Implementación sobre FPGA de un algoritmo de compresión de imágenes hiperespectrales basado en JPEG2000

Daniel Báscones García

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Trabajo de Fin de Máster

Convocatoria: 22 de febrero de 2018
Calificación: 10 - Propuesto a matrícula

Dirigido por:

Carlos González Calvo

Daniel Mozos Muñoz

Autorización de difusión

El abajo firmante, matriculado en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Implementación sobre FPGA de un algoritmo de compresión de imágenes hiperespectrales basado en JPEG2000", realizado durante el curso académico 2017-2018 bajo la dirección de Carlos González Calvo y Daniel Mozos Muñoz en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Daniel Báscones García

Agradecimientos

Tras la carrerilla cogida en el trabajo de fin de grado, me siento en la obligación de salirme de lo convencional para escribir los agradecimientos. Esta sección se ha convertido para mí en una forma de recapitular las experiencias extracurriculares que se han sucedido estos meses, y recordar que no sólo se aprende en el ámbito académico.

Más de dos años han pasado desde aquella reunión de anuncios de TFG de los diferentes departamentos. Para nada esperaba que aquello se acabara convirtiendo en dos artículos y su esencia evolucionara en este TFM. Es verdad aquello de que cuanto más sabes de un tema más pequeño te ves ante la inmensidad del conocimiento. Y todavía me dicen que soy muy alto...

Desde ambos fines del mundo, y desde rincones cercanos. Muchas personas, aun con diferentes destinos, se mueven al mismo ritmo de batallón. Las filas se rompen de vez en cuando, pero vuelven renovadas con fuerza.

Otros están siempre al lado, aun cuando hay distancia de por medio. Consejos, apoyo y guía. Sin duda, para los momentos de duda, la mejor ayuda.

Y cuando hay que hacer paraditas, que sea con tostadas, embutidos, queso o chocolate, ¡qué bien saben! Al igual que el fuego, de vez en cuando hace falta aire fresco para avivar la mente. Playa, montañas, nieve, en coche, en avión, y, al final, volviendo a andar con quienes más importan.

Gracias a los despachos en que se aprovecha el tiempo, a los jefes que no quieren que les llames como tal, a los compañeros que se han ido, a los que siguen, a las cenas de cuatro, a las bibliotecas donde no se estudia, a los guerreros de Uganda, a las pérdidas de tiempo y a los maratones que desgastan la mente. La meta ya está aquí.

Resumen

Las imágenes hiperespectrales son uno de los métodos de análisis disponibles hoy en día para estudios no intrusivos, a distancia, de cualquier elemento.

Capturando cientos de longitudes de onda en cada píxel, la información que proporcionan, además de útil y precisa, es pesada. Una sola imagen puede superar el Giga Byte, por lo que la compresión es más una obligación que una opción.

Si queremos reducir en una fracción importante el tamaño, debemos ceñirnos a los algoritmos con pérdida. En los últimos años han ido evolucionando, con las técnicas más prometedoras utilizando híbridos entre algoritmos de compresión para imágenes tradicionales, y técnicas que incorporan una decorrelación espectral en cada píxel mediante reductores dimensionales.

En este trabajo de fin de máster se ha partido de esa idea, diseñando un compresor con múltiples reductores dimensionales, utilizando como núcleo de la compresión las ideas del estándar JPEG2000, que han sido ampliadas con numerosas opciones. Los resultados muestran niveles de compresión por encima de los obtenidos anteriormente, con la elección de parámetros jugando un papel fundamental en la calidad de las imágenes comprimidas.

A partir de ese desarrollo, se realizó un análisis de tiempos del algoritmo, detectando las partes más lentas. Mediante técnicas de submuestreo en la reducción dimensional, los tiempos se mejoraron sin afectar a la precisión.

Se vio además la posibilidad de conseguir mejoras adicionales incorporando una FPGA como coprocesador para la codificación de JPEG2000. La implementación VHDL ha dado excelentes resultados, y gracias a sus características es arbitrariamente paralelizable, haciendo que la codificación sea prácticamente instantánea en las FPGA con más capacidad del mercado.

Juntando todas estas ideas, se ha conseguido un compresor híbrido capaz de reducir los tiempos de cálculo de varios minutos a meros segundos, posibilitando la compresión con pérdida en tiempo real, a la vez que se mantiene un alto rendimiento de distorsión frente a ratio de compresión.

Palabras clave

Compresión, FPGA, Imagen hiperespectral, JPEG2000, reducción dimensional.

Abstract

Hyperspectral images are one of the latest methods for non-intrusive analysis, allowing distant studies of virtually any subject.

Capturing hundreds of different wavelengths in every pixel they provide valuable and precise, yet heavyweight, information. An image alone can surpass a GigaByte, so compression is no longer optional.

If our goal is to significantly reduce image size, we must focus on lossy algorithms. They have been evolving in the past years, with the most promising techniques being a hybrid between traditional image compression algorithms and spectral decorrelation at the pixel level, using dimensionality reduction.

This master's thesis is based on that idea. A compressor with multiple dimensionality reduction algorithms is designed, with the ideas of the JPEG2000 standard at its core, and is extended with new options. Results show compression ratios higher than those previously obtained, with the choice of parameters playing a crucial role in compression quality.

Based on those findings, the different steps of the algorithm were time profiled to find the slowest bottlenecks. Execution times were greatly improved, without a hit in accuracy, using subsampling techniques in the training steps of the dimensionality reduction algorithms.

It was also found that the coding step of the JPEG2000 algorithm could be significantly sped up on custom hardware. To address this, an FPGA implementation was designed using VHDL, which greatly improved execution times thanks to the parallelizable nature of the algorithm.

Putting all these ideas together, a hybrid compressor is presented which reduces compression time from minutes to seconds, allowing real-time lossy compression with a great rate-distortion performance.

Keywords

Compression, FPGA, Hyperspectral image, JPEG2000, dimensionality reduction.

Índice General

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Seis de septiembre de mil setecientos sesenta y seis. Es la fecha en la que nació John Dalton.

Avanzado estudiante en matemáticas, a los doce años ya daba clases a sus vecinos. Quiso estudiar derecho, pero acabó siendo una figura prominente en química. Como muchos otros genios, destacaba en cualquier campo que rozara (incluso remotamente) su interés.

Su teoría atómica es considerada su mejor aporte a la comunidad científica. En ella [1] postulaba que toda la materia se constituía de unas partículas indivisibles llamadas átomos. Todos los átomos de un elemento eran idénticos. Combinaciones progresivamente complejas de átomos daban lugar a moléculas, compuestos y otras superestructuras.

Pero una vida tan llena de éxitos no vino sin sus dificultades. Dalton, al igual que su hermano, notaba que era diferente a los demás. Le gustaba ser discreto, pero hasta sus vestimentas grises recibían miradas extrañas. Las obras de arte no le transmitían el mensaje que otras personas decían ver.

Por primera vez en la historia [6], Dalton documentó científicamente su problema, y presentó una idea para explicarlo: El humor vítreo (sustancia gelatinosa que rellena el globo ocular) de sus ojos, si bien transparente, contenía pigmentos que absorbían el color rojo y verde, mientras que el resto del espectro, que él sí distinguía correctamente, era detectado por la retina.

Figura 1.1: Siguiendo el orden de lectura: Imagen sin modificar [2], visión con tritanopia [3] (carencia de visión del color azul), visión con deuteranopia [4] (carencia de visión del color verde), y visión monocroma o acromatopsia [5] (solo se distingue luminosidad). Diferentes daltonismos.

Su teoría resultó ser errónea, si bien su exhaustiva investigación hizo que la enfermedad finalmente fuera nombrada Daltonismo. Ciertamente, sus ojos no distinguían los colores rojo y verde, pero no por falta de luz, sino por falta de células receptoras de dichos colores.

¿Qué ocurría? Al igual que los átomos eran los bloques básicos de la materia, también los colores se pueden construir a partir de unos pocos ladrillos. En el ojo, estos ladrillos son denominados conos, y hay tres tipos:

- **L**: Que detectan colores de longitud de onda larga, como el rojo.

- **M**: Encargados de detectar ondas medias, como las del verde, y que son los más sensibles.
- **S**: Para ondas cortas, como las del azul.

Este defecto en el ojo de Dalton le hacía distinguir algo menos de la mitad de colores que una persona normal. En ocasiones resultaba un impedimento para distinguir compuestos químicos en su laboratorio, y tenía que fiarse ciegamente de las etiquetas que decoraban los matraces.

En el extremo opuesto se sitúa el tetracromatismo. Confirmado doscientos años más tarde por Gabriele Jordan y su equipo [7]. En este caso, en lugar de un cono menos, el sujeto tetracromático tiene un cono más. No podemos saber cómo ve un tetracrómata a menos que nosotros mismos lo seamos, pero si Dalton ya comentaba sus dificultades de reconocimiento con la falta de un receptor, podemos imaginar que los tetracrómatas sean para alguien normal lo que alguien normal era a Dalton.

Cualquier fuente de luz, ya sea creada o reflejada, tiene un color particular, dado por la longitud de onda de esta. Normalmente no existen colores "puros" con una única longitud de onda, sino que vemos un continuo de longitudes de onda, denominado espectro, y que caracteriza unívocamente el objeto que la genera, así como los materiales que lo componen.

Este espectro característico es especialmente útil a la hora de distinguir las abundancias de diferentes elementos químicos. Debido a las estructuras de los átomos, cada uno emite luz en ciertas longitudes de onda concretas, y dado el espectro podemos hacer una estimación de qué lo generó.

Un espectrógrafo es un aparato capaz de, dada una fuente de luz, separarla en su espectro. La fabricación del primero se atribuye a Henry Draper [9]. Este aparato analógico producía una imagen del espectro sobre papel fotográfico, y un análisis a mano revelaba los datos que escondía.

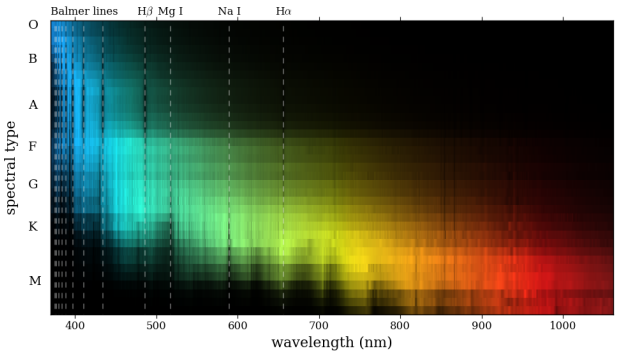


Figura 1.2: Espectro de los diferentes tipos de estrellas [8]. En vertical las líneas correspondientes a los elementos más frecuentes. Se puede conocer el tipo de estrella a partir de su espectro.

Con el avance de la tecnología y el auge de lo digital, se pueden conseguir medidas más precisas del espectro, muestreando longitudes de onda exactas con ordenadores, en lugar de aproximaciones en un papel.

Al igual que un número mayor de conos hace que distingamos más colores, un número mayor de muestras en un espectro aporta más información sobre la fuente de luz.

La evolución de la fotografía comenzó en el blanco y negro, avanzando hasta los tres colores propios de un humano, y saltando a más de tres en el ámbito científico en los últimos años [10-15].

Es precisamente esto lo que llamamos imagen *hiperespectral*. Cada píxel captura cientos de longitudes de onda, ridiculizando la capacidad de distinción de colores que envidiaba Dalton.

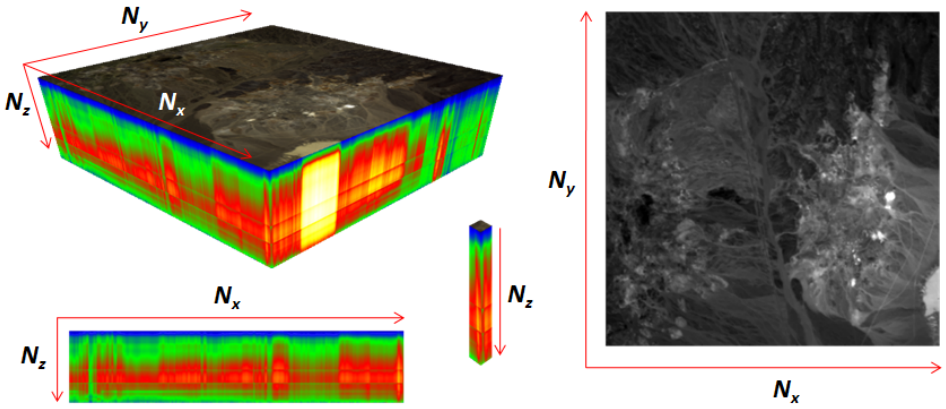


Figura 1.3: Ejemplo de una imagen hiperespectral. Arriba a la izquierda una representación mostrando las tres dimensiones, con el espectro visible proyectado arriba. Abajo, un frame de la imagen (corte vertical), seguido de un *píxel*, y por último una *banda*, que es monocromática.

1.1. Imágenes hiperespectrales

El interés de las imágenes hiperespectrales radica en realizar estudios sobre la composición material del objeto capturado a distancia, sin interactuar físicamente con él. Así, surgen numerosas aplicaciones: En medicina, utilizándose entre otros para detección de tumores [16], en agricultura, para analizar el estado de los cultivos [17], o en ciencias geoespaciales, para monitorizar el estado de la tierra y atmósfera [18].

Una imagen hiperespectral tiene las dos dimensiones espaciales habituales en cualquier imagen (alto y ancho), y añade una dimensión de profundidad, la dimensión espectral (Figura 1.3). Se divide así la imagen en bandas, cada una recogiendo la muestra de una longitud de onda concreta para cada píxel.

Ya estamos familiarizados con el creciente tamaño de las imágenes, cada vez de mayor calidad, que a menudo provocan los mensajes de "espacio insuficiente" en nuestros dispositivos móviles. Esas imágenes únicamente cuentan con tres bandas para satisfacer los tres tipos de conos del ojo.

Aquí el tamaño se multiplica rápidamente con el número de bandas, y más aún teniendo en cuenta que cada muestra (elemento individual) de la imagen tiene usualmente una profundidad de 2 bytes. (frente a 1 que suele utilizarse en fotografía estándar).

Siempre es interesante aprovechar al máximo el espacio del que disponemos, pero más aún en entornos desconectados como puedan ser los aviones o satélites desde los que se sacan algunas imágenes hiperespectrales. Es por tanto útil tener herramientas para hacer una compresión eficiente y rápida de las mismas.

1.2. Compresión

La fábula de la hormiga y la cigarra ya ilustraba las ventajas del ahorrador, persona hoy en día más esquiva que los fugitivos del lejano oeste.

El ahorro es habitualmente visto como la capacidad de almacenar un bien para poder sacar más provecho del mismo en un futuro. Pero el bien que perseguimos acumular en la compresión es el vacío, la nada. Queremos tener menos para poder tener más.

En el mundo analógico de principios de siglo XX, la información se transmitía tal cual se generaba. Un programa de radio era equiparable a un enorme altavoz sonando en la atmósfera, pero con un tipo de onda que sólo se podía escuchar con ayuda de un receptor. Cuentos, historias, o radionovelas se emitían a quien quisiera escuchar.

Y si querías escuchar, lo podías hacer como si estuvieras en la misma mesa del locutor. Podía haber habido distorsiones por el camino, pero en condiciones ideales la calidad era perfecta. Y aún así, la compresión era utilizada en ocasiones. Antes del capítulo del día se podía hacer un resumen del anterior, por si te lo habías perdido por cualquier motivo. La información se seguía transmitiendo sin modificación, pero no se transmitía toda la información.

En el mundo digital cambió el paradigma. Las señales que antes eran analógicas ahora se medían numéricamente, y la exactitud del continuo se perdía en el discreto. Pero esta medida discretizada era el nuevo estándar de perfección.

En este mundo numérico, surgía una nueva posibilidad a la hora de comprimir. Los resúmenes seguían siendo posibles, pero ahora, con una manipulación inteligente de la matemática, podíamos reducir la cantidad de información enviada sin afectar a la calidad de la misma.

Numerosas técnicas que veremos en el Capítulo 3 han ido surgiendo a lo largo de los años, tanto con como sin pérdida. Las técnicas más elaboradas, sin embargo, combinan ideas de ambas, habitualmente decidiendo qué partes de la información son menos importantes, elaborando verdaderos resúmenes de los datos de entrada.

Rodeando a las imágenes hiperespectrales han surgido numerosas técnicas de compresión. Desde opciones sin pérdida [19], para estudios precisos, hasta opciones más permisivas que amplían técnicas ya aplicadas en las imágenes digitales tradicionales. Una técnica muy prometedora, y base de este trabajo fue presentada en [20]. En ella se mezclan técnicas ya probadas del famoso grupo Joint Photographic Experts Group (JPEG) [21] (que da nombre al formato de imagen), con técnicas modernas provenientes del mundo del *machine learning*.

Figura 1.4: Una FPGA soldada en una placa base, que interconecta numerosos periféricos.

1.3. FPGA

La complejidad de algoritmos tan elaborados no es nada despreciable, y procesamientos del orden de varios minutos no son infrecuentes. Es el precio a pagar por tener una compresión fuerte que mantenga la esencia de las imágenes intacta.

La evolución vista por el silicio en los últimos 50 años no tiene precedentes en innovación tecnológica, y una de las hojas de sus múltiples ramas son las Field Programmable Gate Array (FPGA).

Cuando hablamos de programar un procesador, nos viene a la cabeza la idea de concatenar instrucciones de código como si de una receta se tratase, consiguiendo al final que nuestra máquina la ejecute y consiga el resultado esperado. Las características de la máquina pueden hacer que sea mejor o peor para el trabajo que estamos realizando, si bien siempre será capaz de hacerlo.

En una FPGA no programamos las instrucciones, sino que montamos la máquina. Va a hacer una única función, así que únicamente utilizaremos las piezas que nos den el mejor rendimiento posible.

Precisamente esta es la gran ventaja de las FPGA: nos proporcionan un arsenal de opciones difícilmente alcanzable por otros chips, como son:

- Memoria: Si nuestro diseño necesita memoria, la FPGA ya viene con decenas de bancos, e incluso interfaces con RAM como un procesador de propósito general.
- Funciones matemáticas: ¿Necesitamos un multiplicador? ¿Por qué no cincuenta? Las FPGA tienen infinidad de módulos predefinidos.
- Interfaces: Modernas y rápidas como USB y Ethernet, o enfocadas al internet de las cosas como los protocolos I2C o SPI. Cualquiera está disponible.
- Y mucho más: Además de multitud de opciones ya diseñadas, cualquier circuito personalizado puede implementarse, utilizando pequeños bloques reprogramables distribuidos por todo el silicio.

A la hora de programar una FPGA, configuramos todos los recursos, que son conectados mediante una extensa red de interconexiones, que permite la existencia de los más complejos circuitos.

Montar un circuito de este tipo es notablemente más difícil que utilizar una máquina ya montada, por lo que los tiempos y costes de desarrollo con FPGA son mayores que aquellos con procesadores de propósito general. No obstante, los rendimientos conseguidos pueden superar, con mucho, a los tradicionales.

1.4. Motivaciones y objetivos

En este trabajo se unen estos tres pilares: Imágenes Hiperespectrales, compresión y FPGA.

Los algoritmos existentes sin pérdida consiguen buenos resultados de tiempo, pero las tasas de compresión quedan muy lejos de las obtenidas en algoritmos con pérdida.

Por su parte, los algoritmos con pérdida no consiguen un rendimiento tan bueno como sus homólogos del campo de las imágenes tradicionales, y además no están lo suficientemente optimizados en tiempo de ejecución.

Vamos a desarrollar un algoritmo partiendo de la base sentada en [20], que consiga buenos resultados de compresión, tiempo y calidad de imagen. Al algoritmo desarrollado se le aplicarán nuevas ideas del campo de la compresión, ampliando las capacidades del mismo e, idealmente, consiguiendo mejoras sobre el original.

A continuación se analizará el rendimiento, explorando los puntos de mejora tanto en la algoritmia como en la optimización del código a ejecutar. Se buscará diseñar un circuito para FPGA que consiga apoyar al software para mejorar las partes más lentas y paralelizables del algoritmo.

En definitiva, un sistema híbrido que aproveche las mejores características del desarrollo rápido de software, y la capacidad de prueba que ofrece la ejecución sobre un procesador, con la precisión y el rendimiento de la creación de un circuito específico para la compresión en FPGA.

Capítulo 2

Introduction

September the sixth of one thousand seven hundred sixty six. John Dalton is born.

Quite above average in maths, he was already teaching his neighbors at just twelve years old. Inclined at first to study law, he ended up being among the greatest chemists of all time. Like most geniuses, he stood out in every field of knowledge he showed even the slightest interest for.

His atomic theory is considered his greatest scientific achievement. All matter was made up of indivisible building blocks called atoms [1], and all of the atoms of a particular element were identical. Complex combinations of atoms yielded progressively complex structures such as molecules, compounds and other superstructures.

But this life of achievements was not easy. Dalton noticed that he and his brother were different from the rest. He did not like to stand out, but even his dullest clothes raised eyebrows. Art masterpieces didn't fully transmit to him the message others claimed to see.

For the first time in history [6], Dalton scientifically documented his problem, and a hypothesis was made to explain it: His vitreous humor (gelatinous substance that fills the eye), although transparent, contained some pigments that filtered out the red and green colors. His retina was then able to only detect the rest of the visible spectrum.

This hypothesis turned out to be wrong, but thanks to the extensive research done the illness was eventually named after him. It was true that his eyes perceived red and green as the same color, but it was not because of a lack of light, but because of a lack of cells sensitive to those colors.

Figura 2.1: Following the reading order: Raw image [2], tritanopia vision[3] (non sensitive to blue), deuteranopia vision [4] (non sensitive to green), and monochrome vision or achromatopsia [5]. Different types of daltonism.

The same way that combinations of atoms made all matter, all colors could be obtained from just a few select ones. Our vision, for example, relies on just three building blocks, or cones, to resolve all colors:

- **L**: Sensitive to **L**ong wavelength colors, such as red.

- **M**: Used for **M**edium wavelength colors, such as green. These are the most sensitive of all three.
- **S**: They detect **S**hort wavelength colors, like blue.

According to Dalton, his defect made him unable to tell apart about half the colors a normal person could. It was sometimes hard for him to differentiate between chemicals on his lab, and had to blindly trust the tags he or his assistants had put on the glassware.

On the other end of the spectrum is tetrachromatism. Confirmed two hundred years later by Gabriele Jordan's team [7]. A patient with this condition, instead of lacking a cone type, has a fourth one. Us trichromats can never fully grasp the difference with tetrachromats. But with Dalton showing his difficulties because of the lack of a receptor, we can only guess a difference of the same magnitude between tri and tetrachromats.

Every light source or reflector has a particular color, given by its wavelength. A "pure" color with a single wavelength does not normally exist, and is instead a continuum across different wavelengths. This is what we call the spectrum, and it is a signature of the object generating it, as well as the materials it is made out of.

This characteristic spectrum is specially useful for determining the abundance of the different elements. Due to the structure of atoms, each one emits light in certain wavelengths, so given the spectrum we can infer which elements generated it.

An spectrograph is a device capable of splitting a single light source into its spectrum. The first one was made by Henry Draper [9]. This analog device produced an image of the spectrum developed over photographic paper, which was analyzed to show what the light was made of.

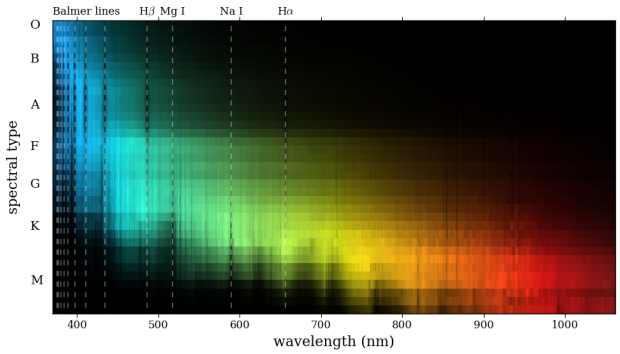


Figura 2.2: Spectra of the different types of stars [8]. Vertical lines show the most common elements. The star type can be inferred from its spectrum.

With technological advancements in the digital era, we can now precisely study the spectrum, sampling exact wavelengths with computers, instead of approximations on paper.

The same way that a higher cone number enables greater color differentiation, a higher number of samples in a spectrograph gives us more information about the light source.

The evolution of photography started in black and white, progressing to the three colors a human can detect, eventually leaping over to quite more than three in scientific experiments [10-15].

This is what we call *hyperspectral* image. Every pixel contains hundreds of wavelengths, overshadowing the minuscule color resolution Dalton wished he had.

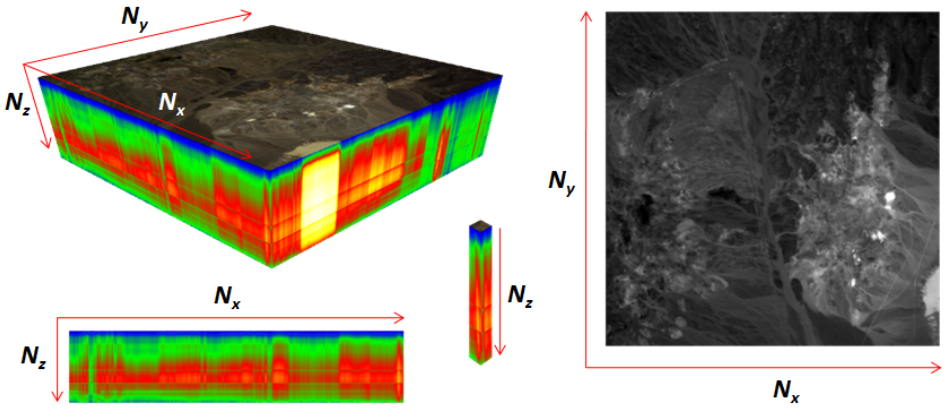


Figura 2.3: An example of a hyperspectral image. Top-left: full image with the visible spectrum projected above. Below, a frame of the image, followed by a *pixel* and lastly a *band*.

2.1. Hyperspectral images

Hypespectral images's most useful application is enabling remote studies of the captured object, with no need of physically interacting with it (apart from lighting it, of course). It has therefore found its way in many fields: Medicine, where it is being used to find tumors [16]. Agriculture, where the health of crops can be monitored [17]. And even geospatial sciences, to study the earth and atmosphere [18].

A hyperspectral image adds a third spectral dimension (Figure 2.3) to the usual spatial dimensions of a normal image (width and height). The image is thus made up of bands, each one with all of the samples of a specific wavelength. A pixel contains then the full spectrum of a specific location, with one sample per band.

We are already familiarized with the increasing image sizes on all of our devices, which are usually behind all of the "not enough storage" messages we often get. Those images only contain information for three wavelengths, one for each cone.

Here, size is quickly boosted with the increasing number of bands, even more so with each sample being a couple of bytes long (on the other hand, only 1 byte is usually enough even for professional photography).

It is always interesting to make the most out of the storage we have available, specially in isolated environments such as planes or satellites, where hyperspectral images are often captured from. Tools to achieve an efficient and fast compression are very interesting in this area.

2.2. Compression

The ant and the grasshopper fable already illustrated the benefits of being a saver.

Saving is usually seen as the ability of storing a good to make the most out of it in the future. But here, what we want more of is storage. We want to have less to be able to store more.

In the analog world of the twentieth century, information was transmitted raw. A radio show was like a giant speaker resounding in the atmosphere with a wave that could only be listened to with the aid of a specific receptor. Anyone could listen to anything that was broadcast.

If you wanted to listen, you could do it just as if you were sat right next to the people speaking. In some situations distortion built up along the way, but in ideal conditions the quality

was perfect. Even then, compression was sometimes used. Just at the beginning of a show, a little summary could be made for the previous one. Information was still fully transmitted, but not all of the information was transmitted.

A paradigm shift was seen when the digital world emerged. Analog signals were now numerically measured, and the precision of the continuum was lost in the discrete. These new discrete measures were now the standard for perfection.

A new way of compressing information was possible now. Apart from summaries, a clever manipulation of numbers could now reduce the amount of information sent, with no effect on its quality.

Throughout the years, multiple techniques, which we'll dive into in Chapter 3, have been developed in the form of lossy and lossless algorithms. The better techniques mix both approaches, usually deciding which bits of information are worth keeping, truly summarizing the input data.

A lot of compression techniques have surfaced along with hyperspectral images. From lossless [19] compression to more relaxed lossy options, which explore and improve compression algorithms already used in traditional imaging. A very promising algorithm, which is the foundation of this work, was presented in [20], where JPEG's (Joint Photographic Expert Group) ideas [21] are combined with dimensionality reduction techniques from the emerging world of *machine learning*.

Figura 2.4: An FPGA soldered in a PCB, which interconnects peripherals to it.

2.3. FPGA

The complexity behind such intricate algorithms is huge, and processing times in the order of minutes per image are the norm. It is the trade-off of having heavy compression while preserving a good image quality.

Computer evolution has been unrivaled for the past 50 years, being the leading force of innovation in technology. FPGAs (Field Programmable Gate Arrays) are one of the most promising branches of this silicon tree.

When we talk of programming a processor, we usually think about chaining simple instructions one after another, like a recipe, eventually executing and obtaining the desired results. Specifics about the processor can make the execution times vary, but in the end the result will be output.

We do not program instructions for an FPGA. Instead, we assemble the machine from scratch. A circuit is designed for a specific purpose, which will be implemented over a fabric of logic gates and interconnection elements.

This is precisely the great advantage over traditional silicon chips. FPGAs offer multiple options hardly available in other platforms, such as:

- Memory: If our design is memory hungry, our FPGA can use its memory banks, or even interface with RAM just as a general purpose processor would do.

- Math Functions: Do we need a multiplier? Why not fifty? The latest FPGA come with thousands of modules ready to compute any mathematical operation.
- Interfaces: Fast like USB and Ethernet, or Internet-of-things focused like the more light-weight protocols I2C or SPI. Virtually any interface is available.
- And many more: Aside from predefined options, any custom circuit can be implemented, using the re-programmable blocks that populate the silicon fabric.

When programming one of these beasts, we configure which resources we are going to use, how we are going to use them, and how they connect to each other. All of this allows for the implementation of the most simple circuits, up to fully blown processors.

Putting all of this together is notably harder than using an already assembled machine, so development costs and times are higher than when using general purpose processors. However, performance can improve by quite a bit, so it is an interesting path to explore in some situations.

2.4. Objectives and motivations

This work is built on three pillars: Hyperspectral images, compression and FPGAs.

Existing lossless algorithms are fast, but compression rates are miles away from those of their lossy counterparts.

These lossy hyperspectral algorithms are not as optimized as the lossy algorithms for photographs, which have been refined far more due to their demand.

Here we develop an algorithm based on the ideas from [20], aiming for good results in compression rate, execution time and image quality. We will employ different compression techniques, broadening the algorithm's capabilities and ideally improving the original.

The performance will be analyzed, exploring which parts can be improved both from a theoretical algorithmic standpoint as well as a practical implementation one. We will introduce FPGAs as co-processors in the compression flow, helping in the slowest and more parallelizable parts of the algorithm.

The result is a hybrid system that benefits from quick software testing, as well as from the performance that results from creating a specific circuit aimed at compression using FPGAs.

Capítulo 3

Compresión

Cada vez tenemos más espacio en nuestros dispositivos electrónicos. Ya hace décadas que dejamos de hablar en "kas" y "megas", y los "gigas" recientemente han sido sustituidos por los "teras". Con este aumento exponencial de la capacidad, uno podría preguntarse qué sentido tiene comprimir la información. Total, lo que antes suponía varias estanterías de álbumes, libros, cassettes y cintas de vídeo, ahora cabe sin problemas en la palma de la mano.

Aunque tendamos a atribuir este mérito a los avances físicos de la tecnología, gran parte del mismo corresponde a avances en el plano teórico de *cómo* guardamos la información. Pongamos como ejemplo un vídeo cualquiera de nuestro móvil, de los cuales el cuñado de turno envía varios al día. En resolución 1920x1080 (full HD), a 24 fotogramas por segundo, y con color Red Green Blue (RGB) de 24 bits, un vídeo de 15 segundos sin comprimir ocupa la friolera de $2{,}24GB$.

Es aquí donde comenzamos toda la maquinaria oculta que hay tras el telón, que hace que los $64GB$ de un móvil nuevo no se llenen con una veintena de vídeos de las vacaciones de la familia. En el caso de los vídeos, el último grito en compresión, el estándar H.265 hace que una película pase de ocupar casi un "tera" a tan solo unos pocos "gigas".

Pero algo tan bueno debe tener algún pero. Este tipo de compresión tan drástica introduce pequeñas distorsiones en los datos originales, y los algoritmos que la llevan a cabo suelen tardar un tiempo considerable en completarla. Aun así, las distorsiones suelen ser imperceptibles, y el tiempo de compresión es más que compensado en cuanto tengamos que enviar el archivo un par de veces por internet.

Y precisamente es internet el gran propulsor de las tecnologías de compresión. Más concretamente, las empresas que manejan cantidades ingentes de datos. A nivel personal comprar un disco duro más o menos no supone un gran desembolso. Sin embargo, empresas de almacenamiento en la nube, emisiones en directo, o vídeo bajo demanda sí están interesadas en reducir al máximo la información que guardan o transmiten. Porque no es lo mismo que el usuario reciba el último capítulo de su serie favorita en un minuto que en una hora, y no es lo mismo mantener un edificio lleno de servidores que dos.

3.1. Orígenes - Teoría de la información

Fue Shannon en 1948 [22] quien sentó las bases matemáticas de la teoría de la información. Con el concepto de entropía puso un límite teórico a cuánta información se podía transmitir por un canal. Teniendo cierto conocimiento previo de la fuente de información (posibles símbolos a transmitir, así como sus frecuencias), es posible enviar mensajes con una menor cantidad de datos que si tuviéramos que asumir una fuente aleatoria, utilizando códigos cuidadosamente formados.

Definición 1 *Un **código**[1] es un diccionario que asocia una serie de símbolos a codificar de un alfabeto A de entrada con símbolos de otro alfabeto B que es el que soporta el canal de comunicación.*

Por ejemplo, si A es el abecedario, y B el conjunto de símbolos binarios, un posible código es:

$$C = \{a \to \texttt{00000}, b \to \texttt{00001}, \ldots, z \to \texttt{11000}\}$$

*La **longitud** de un código es la longitud de las cadenas mapeadas de símbolos de B, en este caso 5. Los códigos pueden tener diferentes longitudes para diferentes símbolos, en cuyo caso son **de longitud variable**.*

Consideremos una variable discreta aleatoria X que emite símbolos $x_i, i \in [1, n]$. Para codificarlos podemos utilizar binario, con códigos de $L(X) = \lceil \log_2(|X|) \rceil$ bits, que acomodan unívocamente todos los posibles símbolos. En determinados casos, las probabilidades de aparición de cada símbolo $P(x_i)$ pueden estar sesgadas, y entonces la forma más eficiente de codificarlos es asignar a cada uno un código de longitud variable, donde los símbolos más frecuentes utilizarán menos bits, y los más infrecuentes cadenas más largas. La longitud media con la que se transmiten estos símbolos tiene un límite teórico conocido como entropía de Shannon ($H(X)$).

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2(P(x_i)) \tag{3.1}$$

Donde la base del logaritmo es el número de elementos en el alfabeto que codifica los x_i, en este caso 2 por utilizar binario.

Se cumple que $H(X) \leq L(X)$, y por tanto la codificación trivial puede ser usualmente mejorada con una mejor elección de los códigos. Un simple ejemplo se encuentra en el trabajo de Shannon: Codificamos símbolos A, B, C, D con probabilidades $1/2, 1/4, 1/8, 1/8$ respectivamente. En este caso $L = \lceil \log_2(4) \rceil = 2$, y sin embargo:

$$H(X) = -\left(\frac{1}{2}\log_2\left(\frac{1}{2}\right) + \frac{1}{4}\log_2\left(\frac{1}{4}\right) + 2\frac{1}{8}\log_2\left(\frac{1}{8}\right)\right) = \frac{7}{4} \tag{3.2}$$

Por tanto vemos que, codificando 2 bits por símbolo, desaprovechamos parte de los datos transmitidos, que son redundantes. De hecho, empleando el código $A \to \texttt{0}$, $B =\!\!\to \texttt{10}$, $C =\!\!\to \texttt{110}$ y $D =\!\!\to \texttt{111}$, conseguimos que la media de bits transmitidos alcance el límite teórico, pues para N símbolos, se transmiten:

$$bits(N) = N\left(\frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \frac{2}{8}\cdot 3\right) = \frac{7}{4}\cdot N \tag{3.3}$$

Este límite no siempre es alcanzable. Por ejemplo si $n = 2$, el mejor resultado que podemos conseguir es un bit por símbolo, asignando 0 y 1 indistintamente a x_1 y x_2. La entropía sin embargo puede ser arbitrariamente cercana a cero si sesgamos la probabilidad $p = P(x_1)$:

$$\lim_{p\to 1}(H(X)) = \lim_{p\to 1} -(p\log_2(p) + (1-p)\log_2(1-p)) = 0 \tag{3.4}$$

Un código que codifique cada símbolo con el mismo número de bits n es fácilmente decodificable: basta dividir la entrada en trozos de n bits y mirar a qué símbolo corresponden. Sin

[1] Se utilizará código indistintamente para denotar al mapa completo y a la traducción de un único símbolo.

embargo, cualquier código de longitud variable tiene que ser unívocamente decodificable [23] para asegurar que el mensaje no pueda ser interpretado de maneras diferentes.

Un ejemplo de código no unívocamente decodificable es el siguiente [24]:

$$a \to \texttt{1}, b \to \texttt{011}, c \to \texttt{01110}, d \to \texttt{1110}, e \to \texttt{10011} \tag{3.5}$$

En este caso, la cadena **011101110011** puede ser interpretada como **01110-1110-011** (*cdb*) o **011-1-011-10011** (*babe*).

Dispuesto a resolver estos problemas apareció David Huffman, que idearía la manera de crear códigos prefijos con una entropía igual, o muy próxima, a $H(X)$.

Definición 2 *Un **código prefijo** es un código en el cual ninguna representación de un símbolo es prefijo de otra. Por ejemplo, el código:*

$$C = \{a \to \texttt{11}, b \to \texttt{110}, c \to \texttt{0}\}$$

No es un código prefijo ya que la representación de a es prefijo de la de b.

3.2. Codificación de Huffman

Shannon había dado el límite teórico de cuánto podría reducirse la información en caso de existir una codificación óptima. Estaba claro que en ocasiones existía y en otras no, pero no había un método para generarla en los casos afirmativos.

Huffman creó un algoritmo [25] para encontrar esta codificación óptima (de existir) y si no, para encontrar la mejor posible:

Los símbolos son ordenados de mayor a menor número de apariciones en una lista. Los dos con menor número se agrupan, y se vuelven a añadir a la lista, de manera ordenada, como un único símbolo cuyo número de apariciones es la suma de las apariciones de los que lo formaron. Se procede así hasta que todos los símbolos han sido agrupados, y el único elemento de la lista (que ahora es el conjunto de todos los símbolos) tiene la totalidad de las apariciones.

Nótese la equivalencia entre ordenar por número de apariciones y por probabilidad (que es como originalmente se propone en [25]. Si denotamos T al total de apariciones, y t_i y t_j a las apariciones de símbolos x_i y x_j, tenemos:

$$t_i \geq t_j \Leftrightarrow \frac{t_i}{T} \geq \frac{t_j}{T} \equiv P(x_i) \geq P(x_j) \tag{3.6}$$

Con este método se forma un árbol binario (ver Figura 3.1) cuya raíz es el conjunto de todos los símbolos, y cuyas hojas son los símbolos individuales. En cada nodo, uno de los hijos se etiqueta con 1 y el otro con 0. El código de un símbolo es la concatenación de los símbolos encontrados desde la raíz hasta llegar a la hoja que lo representa.

3.3. Codificación aritmética

El límite teórico sobre cuánto podemos comprimir la información impuesto por Shannon no siempre es alcanzado con la codificación de Huffman. La codificación aritmética, técnica posterior en el tiempo, sí asegura el alcance del límite.

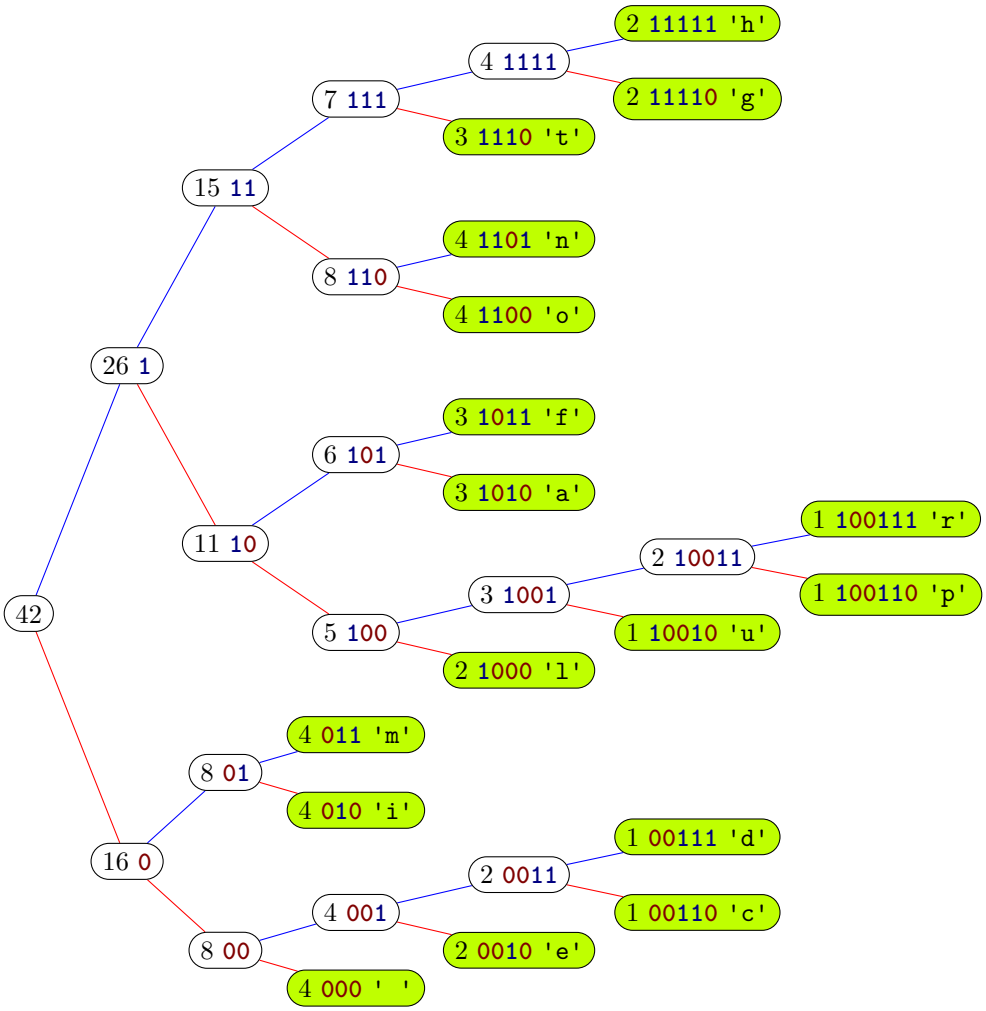


Figura 3.1: Árbol de Huffman para la frase “Implementation of Huffman Coding algorithm”. Las hojas del árbol contienen los códigos empleados para codificar cada carácter. El número de cada nodo es el total agregado de elementos que caen bajo ese prefijo.

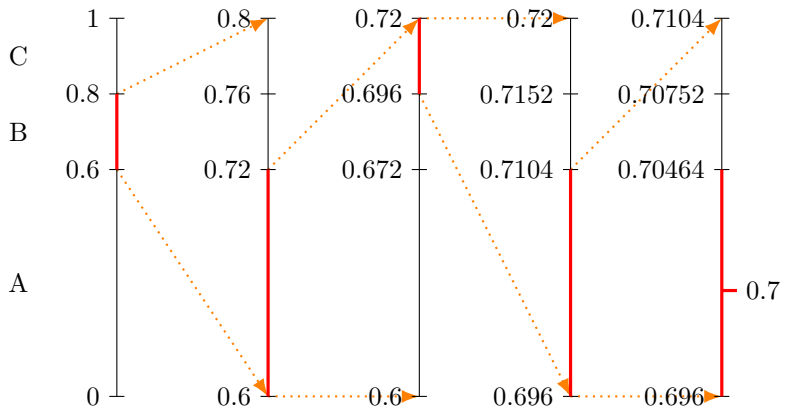


Figura 3.2: Codificación aritmética de la secuencia `BACAA` suponiendo unas probabilidades $P(A) = 0{,}6$; $P(B) = 0{,}2$; $P(C) = 0{,}2$. Sólo necesitaríamos codificar el número de veces que hemos subdividido el intervalo, y una fracción de dicho intervalo. Por ejemplo 5 y 0,7 en este caso.

La idea original [26] se basa en una demostración [27] del primer teorema de Shannon utilizando subdivisiones sucesivas del intervalo unidad.

En la codificación aritmética, un mensaje se representa por un intervalo $[c_n, c_n + a_n) \subseteq [0, 1)$. De dicho intervalo se toma un representante $p \in [c_n, c_n + a_n)$, que codifica nuestro mensaje. Cuanto más largo es el mensaje, más se reduce el intervalo, y el número de bits necesarios para representar p crece. Los símbolos del mensaje van reduciendo sucesivamente el intervalo (ver Figura 3.2) según sus probabilidades de aparición, que son dictaminadas por un modelo. Los símbolos más probables reducen poco el intervalo, añadiendo por tanto menos bits a la representación total del mensaje.

Para actualizar el intervalo dependemos de funciones que nos indican la probabilidad de que aparezca un símbolo x_i de la secuencia a codificar:

- $f_X(x_i)$ que nos dice la probabilidad p_{x_i} de que aparezca x_i en nuestro modelo.
- $F_X(x_i) = \sum_{j=0}^{i-1} f_X(x_j)$ que nos dice la suma de las probabilidades de todos los símbolos anteriores. Si $X = \{x_0, \ldots, x_n\}$, entonces $F_X(X) = 1$.

Con esto, definimos las funciones de actualización del intervalo:

$$a_{n+1} \leftarrow a_n f_X(x_n) \tag{3.7}$$
$$c_{n+1} \leftarrow c_n + a_n F_X(x_n) \tag{3.8}$$

Cualquier $p \in [c_n, c_n + a_n)$ es válido como representación de la cadena codificada. Para decodificar basta volver a dividir el intervalo según el modelo, e ir tomando sucesivamente los símbolos representados por los intervalos a los que pertenece p. Necesitamos también saber el número de símbolos que vamos a decodificar, ya que dada cualquier fracción, podríamos subdividir indefinidamente el intervalo. Esto introduce un pequeño sobrecoste (por ejemplo un número al inicio indicando la cantidad de símbolos a decodificar) negligible cuando codificamos secuencias lo suficientemente largas.

Existe un problema práctico con esta definición, y es que necesitamos precisión decimal arbitraria para guardar y actualizar los límites del intervalo.

Durante un tiempo los codificadores aritméticos no fueron utilizados debido a esta limitación, hasta que se publicó una implementación completa en un artículo [28]. Un par de trucos eran necesarios para que funcionase:

- La implementación tenía que tratar con números enteros para simplificar el proceso de cálculo, por lo que las funciones de probabilidad $f_X(x_i)$ pasaron a aproximarse con enteros de P bits, siguiendo la fórmula:

$$f_X(x_i) \approx p'_{x_i} = \lfloor 2^P p_{x_i} \rfloor \tag{3.9}$$

  Con cuidado de que ninguna probabilidad se redondee a cero.

- Los registros a y c debían ser de longitud finita, por lo que se limitó su tamaño a enteros de N y $N + P$ bits, respectivamente.

Mediante manipulaciones ingeniosas de los registros, se conseguía una cadena totalmente decodificable, si bien en cada momento sólo una fracción parcial estaba disponible. Además, no eran necesarios valores muy altos de N y P para conseguir un resultado aceptable, y manteniendo $N + P < 32$ bits el rendimiento se mantenía cercano al teórico empleando precisión arbitraria. Una explicación muy buena de la implementación se puede encontrar en [29].

3.3.1. Codificación aritmética binaria

Aún se puede simplificar más la idea, utilizando un alfabeto binario de símbolos. El codificador aritmético definido trabaja con alfabetos de cardinal arbitrario. Sin embargo, podemos ver que un codificador binario puede ser totalmente equivalente.

Sin pérdida de generalidad, supongamos que el alfabeto de entrada A_X (donde tenemos que $x_i \in A_X$) tiene una cardinalidad $|A_X| = 2^K,\ K \in \mathbb{N}$. Cada elemento puede representarse por un entero de K bits.

X es una variable aleatoria con 2^K posibles resultados, y podemos interpretarla también como un vector de K variables aleatorias binarias $B_0, \ldots, B_{K-1}$, donde cada B nos proporciona los dígitos binarios de X.

Si en el caso general codificábamos un símbolo x_i en base a su probabilidad p_{x_i}, ahora codificamos K bits con sus K probabilidades correspondientes:

- Para el primer bit solo existe una distribución de probabilidad, que nos indica la probabilidad de que x_i comience por `0` o `1`.

- Para el segundo existen dos, condicionadas por que el primer bit fuera `0` o `1`, y cada una nos indica la probabilidad de que x_i tenga su segundo bit `0` o `1`.

- Continuamos así sucesivamente, hasta ver que el número de distribuciones de probabilidad que necesitamos saber asciende a 2^{K-1} para el último bit. El total de distribuciones es de $1 + 2 + \cdots + 2^{K-1} = 2^K - 1$. Al ser binarias, basta mantener un valor para tener la distribución completa, por tanto el total de probabilidades mantenido es exactamente igual al que teníamos que mantener para saber las probabilidades de cada x_i (sabiendo que $f_X(x_n) = 1 - F_X(x_n)$).

Recordemos que cuando codificábamos elementos $a_i \in A_X$, la entropía, o menor número de bits que era posible utilizar venía dada por H(X). Se cumple aquí que

$$H(X) = H(B_0) + H(B_1|B_0) + \cdots + H(B_{K-1}|B_0, \ldots, B_{K-2}) \tag{3.10}$$

Codificar un elemento x_i de probabilidad p tenía un coste de aproximadamente $-p \log_2(p)$ bits, y si denotamos x_i^j al bit j-ésimo de x_i, tenemos que, siguiendo la Ecuación (3.10), se cumple:

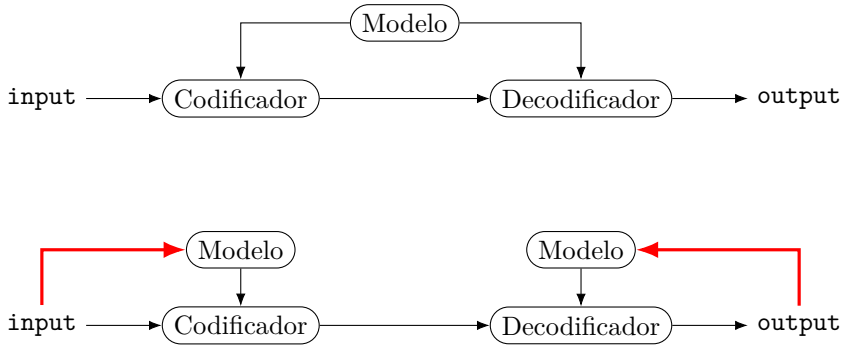


Figura 3.3: En el primer diagrama se muestra un codificador no adaptativo. El modelo es estático y compartido en todo momento por codificador y decodificador. En el segundo diagrama observamos un codificador adaptativo. El modelo se actualiza en cada caso con la entrada o salida. Coincide cuando codificador y decodificador están sincronizados (se han codificado los mismos elementos que se han decodificado).

$$\begin{aligned} -p\log_2(p) = &- p\left(x_j^0\right)\log_2\left(p\left(x_j^0\right)\right) - p\left(x_j^1|x_j^0\right)\log_2\left(p\left(x_j^1|x_j^0\right)\right) - \dots \\ &- p\left(x_j^{K-1}|x_j^{K-2},\dots,x_j^0\right)\log_2\left(p\left(x_j^{K-1}|x_j^{K-2},\dots,x_j^0\right)\right) \end{aligned} \tag{3.11}$$

Y por tanto el coste es exactamente igual si tenemos cuidado de generar correctamente B_i.

Una ventaja de los codificadores aritméticos binarios es que algunas de las distribuciones referentes a los bits menos significativos son muy uniformes, y por tanto podemos asignar $p = 0{,}5$ tanto para `0` como `1` en esas B_i. En este caso necesitaremos aún menos información que para el codificador aritmético clásico, con un rendimiento muy parecido.

3.4. Codificación entrópica adaptativa

Tanto la codificación de Huffman como la codificación aritmética pertenecen a la familia de algoritmos de codificación entrópica. Se basan en tener conocimiento previo de la distribución de probabilidad de los símbolos a fin de codificarlos con resultados lo más cerca posible del límite teórico de Shannon.

Un conocimiento *global* de la información no siempre da los mejores resultados. Por ejemplo, una cadena de la forma 000001111110 tiene una entropía de $1bit$. Sin embargo, si la partimos a la mitad, obtenemos dos cadenas 000001 y 111110 mucho más fáciles de codificar, ambas con entropía 0,65. Por tanto, un método que se adapte a las estadísticas locales funcionará mejor en este caso.

Para construir un codificador adaptativo (ver Figura 3.3) lo único que es necesario es cambiar el modelo de probabilidad de un codificador no adaptativo en tiempo de ejecución. El truco está en que el decodificador también debe ser capaz de reconstruir este modelo. Por tanto el modelo, en un instante j, sólo puede ser función de los símbolos $x_i, i < j$. Por ejemplo, en el caso de un codificador de Huffman, los códigos podrían cambiar cada cierto tiempo para adaptarse a las estadísticas locales ya visitadas. Como el decodificador también va a conocer el estado en ese punto, sabrá cómo leer los nuevos símbolos.

Con estas técnicas es posible vencer el límite de entropía, y comprimir aún más la información sin incurrir en pérdida.

3.5. Codificación de carrera

La codificación de carrera busca hacer resúmenes muy cortos de secuencias largas de símbolos repetitivos. Dada una secuencia de símbolos $S = \{s_1, \ldots, s_n\}$, la codificación de carrera genera una nueva secuencia $S' = \{(n_0, s'_0), \ldots, (n_m, s'_m)\}$ de parejas (n, s), donde n indica el número de veces que hay que repetir s para reconstruir la secuencia original. Un ejemplo de codificación de carrera es:

$$aaaaaabbccccaaaa \rightarrow (6, a)(2, b)(4, c)(4, a)$$

Es claro ver que secuencias con muchas repeticiones pueden resumirse mucho, mientras que otras más aleatorias incrementarán su tamaño al usar esta codificación, debido al sobrecoste de incluir el número de repeticiones junto a cada símbolo.

La codificación de carrera suele utilizarse como apoyo a otros algoritmos, que detectan zonas muy uniformes en los datos, y mediante indicadores especiales señalan el uso de esta codificación en tramos concretos del flujo de datos.

3.6. Reducción de dimensionalidad

Una técnica muy popular [30] a la hora de resumir, visualizar y analizar datos es la reducción de dimensionalidad. Pongamos como ejemplo el reconocimiento de imágenes [31]. Habitualmente tendremos imágenes con millones de píxeles que queremos clasificar de manera discreta en un número no tan grande de clases diferentes. Los píxeles individuales de la imagen no nos aportan información detallada sobre la misma. Lo que define al objeto retratado son una serie de características que podemos inferir a partir de los píxeles (forma del objeto, brillo, color general...). La reducción dimensional busca extraer estas características para hacer más fácil el trabajo con los datos a estudiar.

Matemáticamente, comenzamos en el espacio $\mathbb{R}^n$, y realizamos una transformación al espacio $\mathbb{R}^m$, siendo la dimensión $m < n$. Los datos de entrada son vectores de números reales n dimensionales, que se proyectan a vectores m dimensionales sobre el espacio destino. Cualquier análisis o estudio a realizar siempre será menos costoso computacionalmente en el destino, debido a que trataremos con menos información.

Al hacer reducción dimensional es inevitable perder información. Lo esperado es perder información redundante, pero siempre tendremos cierta pérdida de información útil. Es por tanto necesario saber qué precisión necesitamos para nuestro propósito, y ver hasta dónde podemos explotar la reducción.

En concreto, para imágenes hiperespectrales, existe una gran correlación entre los diferentes píxeles de la imagen. El número de materiales distintos en la escena puede ser de varias decenas, pero no necesariamente alcanza el número de bandas de la imagen. Siendo los píxeles combinación de los espectros de los diferentes materiales, es evidente que con una menor dimensión podríamos representar la misma cantidad de información: cada píxel como combinación de los espectros base.

Una reducción dimensional puede interpretarse como una función $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ que transforma un vector (píxel) $\boldsymbol{x} \in \mathbb{R}^n$ a otro vector $\boldsymbol{t} \in \mathbb{R}^m$. Para hacer el camino de vuelta, se suele generar simultáneamente una función $g : f(\mathbb{R}^n) \subset \mathbb{R}^m \rightarrow \mathbb{R}^n$.[2] Lo ideal sería conseguir que $g \equiv f^{-1}$, pero habitualmente no será posible pues f no será inyectiva sobre su imagen, por

[2]Fuera de la imagen de f no necesitamos siquiera que g esté definida, pues no recuperamos esos valores.

tanto tendremos que conformarnos con una aproximación $g(\boldsymbol{t}) = \boldsymbol{x}' \approx \boldsymbol{x}$. Dicha aproximación será mejor cuanto menor sea $m - n$, y nos hará perder más información cuando $m \ll n$.

Los datos originales los representaremos por la matriz:

$$X = [\boldsymbol{x_1}, \ldots, \boldsymbol{x_p}] \in \mathcal{M}_{n \times p} \tag{3.12}$$

Siendo p el número de muestras. Los datos transformados se representan mediante:

$$T = f(X) = [f(\boldsymbol{x_1}), \ldots, f(\boldsymbol{x_p})] = [\boldsymbol{t_1}, \ldots, \boldsymbol{t_p}] \in \mathcal{M}_{m \times p} \tag{3.13}$$

Para comprimir, lo que haremos será guardar T, así como la función g para recuperar los originales. El sobrecoste de guardar g en general será compensado por la reducción en n/m veces del resto de la información. Veamos a continuación algunos métodos estudiados en este trabajo para reducir la información.

3.6.1. Análisis de componentes principales (PCA)

El Análisis de componentes principales (PCA) es una técnica de reducción dimensional ampliamente utilizada en machine learning [32] por su sencillez, entre otros para visualización de datos o extracción de características latentes [33] en un conjunto de datos.

La definición por la que nos vamos a regir es la de Hotelling [34], donde el PCA es la proyección ortogonal de los datos de dimensión n en un espacio de dimensión menor m, conocido como el subespacio principal, donde la varianza de la proyección se maximiza.

PCA, por tanto, pretende mantener la mayor cantidad de varianza de los datos de entrada, con la esperanza de que así se conserve la mayor cantidad de información posible. Podemos ver un ejemplo donde se aplica PCA de 2 a 1 dimensión en la Figura 3.4.

¿Cómo se calcula PCA?. Para hacer la proyección necesitamos vectores $\{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m\}, \boldsymbol{w}_i \in \mathbb{R}_n, \boldsymbol{w}_i \perp \boldsymbol{w}_j \forall i \neq j$. Estos vectores se juntan en una matriz $W \in \mathcal{M}_{n \times m}$, de tal forma que $f(\boldsymbol{x_i}) = \boldsymbol{x_i} W$, y $g(\boldsymbol{t_i}) = \boldsymbol{t_i} W^T$ definen las funciones de transformación $f : \mathbb{R}^n \to \mathbb{R}^m$ y recuperación $g : \mathbb{R}^m \to \mathbb{R}^n$.

Es claro ver que, en caso de que $W^{-1} = W^T$, entonces $f \equiv g^{-1}$, pero esto sólo es posible si:

- W es ortogonal, que lo es por definición.
- $n = m$, que si nuestro objetivo es reducir información, no será nunca cierto.

Si bien en la práctica, como el dominio de f es un subconjunto de $\mathbb{R}^n$ que no tiene por qué tener dimensión n, es posible que g sí sea la inversa sobre la imagen aun siendo $n < m$.

PCA funciona mejor con datos de media cero. Por tanto se suele calcular:

$$\bar{\boldsymbol{x}} = \frac{\sum_{i=1}^{P} x_i}{P} \tag{3.14}$$

Y trabajar con PCA sobre el conjunto $Z = X - \bar{\boldsymbol{x}}$.

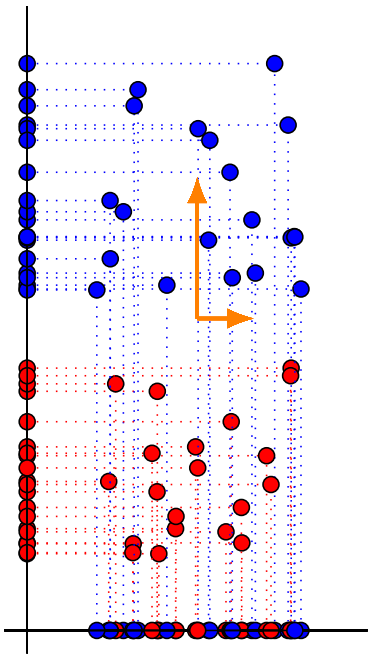


Figura 3.4: Funcionamiento de PCA. En naranja los autovectores. La mayor varianza se da en la dirección del autovector de mayor autovalor.

Para obtener W, en primer lugar se calcula la matriz de covarianza $S \in \mathcal{M}_{n\times n}$ de los datos de entrada $X \in \mathcal{M}_{n\times p}$, siendo p la cantidad de puntos en el espacio de partida:

$$S = \frac{(X - \bar{\boldsymbol{x}})(X - \bar{\boldsymbol{x}})^T}{p-1} = \frac{ZZ^T}{p-1} \tag{3.15}$$

A continuación, se calculan los autovalores y autovectores de S, y se ordenan los autovectores según su autovalor asociado. Los m autovectores con mayor autovalor asociado forman la matriz W. Esta forma de construir W garantiza que proyectar mediante dicha matriz mantiene la máxima varianza posible en la dimensión elegida [32, Ch.12].

3.6.1.1. SVD

La descomposición singular de valores (SVD) es un método similar a PCA que básicamente obtiene el mismo resultado salvo ciertos factores multiplicativos [35]. De hecho en ocasiones se utiliza para calcular PCA debido a sus propiedades numéricas. SVD busca una descomposición de la forma:

$$X^T = U\Sigma V^T, \quad U \in \mathcal{M}_{p\times p}, \quad \Sigma \in \mathcal{M}_{p\times n}, \quad V \in \mathcal{M}_{n\times n} \tag{3.16}$$

Donde tanto U como V son matrices ortogonales, cuyas columnas son respectivamente las direcciones principales izquierdas y derechas, y Σ es una matriz diagonal con los valores singulares σ_i en su diagonal.

La matriz de covarianza S, por ser simétrica, se puede diagonalizar:

$$S = W\Lambda W^t \tag{3.17}$$

Donde W contiene los autovectores como columnas, y Λ los autovalores λ_i de S en la diagonal.

Sustituyendo en la Ecuación (3.15), asumiendo que $\bar{\boldsymbol{x}} = 0$, obtenemos:

$$S = \frac{(U\Sigma V^T)^T U\Sigma V^T}{p-1} = \frac{V\Sigma U^T U\Sigma V^T}{p-1} = V\frac{\Sigma^2}{p-1}V^T \tag{3.18}$$

Por lo que vemos que V nos da vectores proporcionales a W escalados por $\sigma_i/\sqrt{p-1}$, siguiendo la ecuación $\lambda_i = \sigma_i^2/(p-1)$.

Nótese que esto solo es cierto cuando los x_i están centrados con media cero, que es cuando se cumple $S = X^t X/(p-1)$.

Cuando $\bar{\boldsymbol{x}} \neq 0$, la utilidad de PCA y SVD disminuye drásticamente como reductores de dimensionalidad, por tanto en nuestro caso asumiremos que la condición se da, y por tanto utilizaremos uno u otro según cuál tenga mejores propiedades numéricas: bien resolviendo directamente el sistema para S, o calculando la descomposición $U\Sigma V^T$.

3.6.2. Análisis de componentes independientes (ICA)

El análisis de componentes independientes (ICA) es una técnica que busca una representación de un conjunto de datos n dimensional en otro m dimensional ($m \leq n$) de forma que las componentes transformadas sean tan independientes entre sí como sea posible.

Para ello se minimiza la información mutua [36] de las componentes transformadas. La ventaja frente a otros métodos es que ICA es invariante bajo escalado de las componentes (PCA por ejemplo no cumple esta propiedad). Esto es una gran ventaja en el caso de que las magnitudes de las componentes latentes sean muy diferentes, ya que podrá descubrir incluso aquellas que menos aportan al total. Sin embargo puede ser un problema ante la presencia de ruido, ya que puede darle la misma importancia que a una señal fuerte, aun siendo este imperceptible.

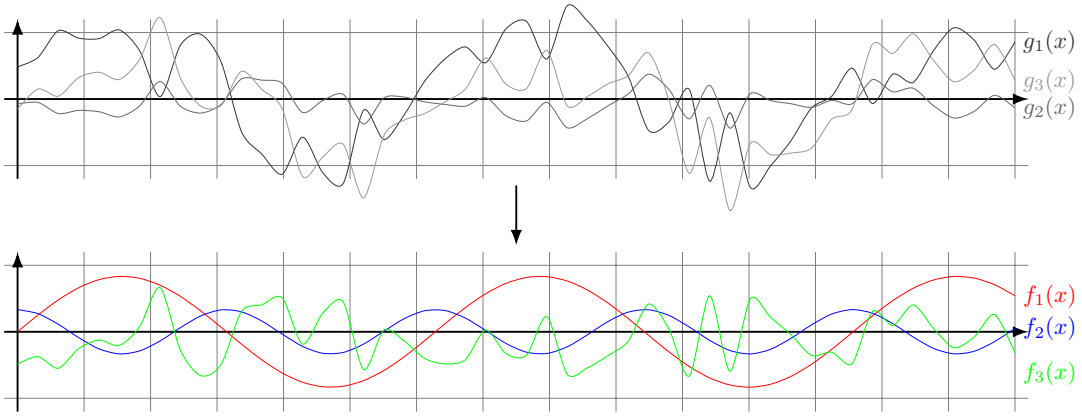


Figura 3.5: Partimos de datos generados con las funciones $f_1(x) = \sin x, f_2(x) = \cos 2x$ y $f_3(x)$ que es ruido pseudoaleatorio. Se generan g_1, g_2, g_3 como combinación lineal de las anteriores. ICA es capaz de obtener, partiendo de los datos mezclados, las componentes subyacentes que los formaban.

Existen diferentes aproximaciones para calcular la información mutua, cuyos resultados se adaptan mejor a datos con unas u otras características. Debido a la naturaleza iterativa de ICA, se suelen utilizar funciones con un coste computacional bajo, con la contraprestación de tener que hacer ciertas suposiciones en los datos de entrada [37].

Utilizado como técnica de reducción de dimensionalidad, ICA asume que existen m componentes independientes latentes[3] en el espacio $\mathbb{R}^n$ origen. Es conveniente que m aproxime con exactitud al número verdadero o de componentes latentes. Errores por exceso implican que se puede detectar ruido como una componente independiente, a menudo indistinguible de componentes de señal. Errores por defecto implican que componentes independientes colapsarán al mismo punto en la dimensión reducida, de nuevo reduciendo el rendimiento.

En cualquier caso, si $m \approx o$, y en particular si $m = o$, el comportamiento del algoritmo es excelente, como muestra la Figura 3.5 para el caso $n = m = p = 3$.

3.6.3. Fracción mínima de ruido (MNF)

La fracción mínima de ruido [38] (MNF) es un método de reducción dimensional similar a PCA, pero que en lugar de ordenar las componentes según la varianza que conservan, las ordena según la cantidad de ruido que contienen. Para ello, no sólo opera sobre la matriz de covarianza de las muestras S, sino además sobre la matriz de covarianza del ruido $S_\mathcal{N} \in \mathcal{M}_{n\times n}$.

$S_\mathcal{N}$ generalmente no es conocida de antemano, por lo que se puede estimar [39] si suponemos correlación espacial entre los x_i cercanos. Esto es razonable ya que en una imagen *natural*, píxeles cercanos tienden a ser parecidos. Para ello estimamos el ruido en cada punto creando la matriz de ruido $\mathcal{N} = [n_1, \ldots, n_p] \in p \times n$, siguiendo por ejemplo:

$$n_i = \frac{1}{2}(x_i - x_{i+1}) \tag{3.19}$$

La matriz de correlación de ruido es entonces:

$$S_\mathcal{N} = \mathcal{N}\mathcal{N}^T \in \mathcal{M}_{n\times n} \tag{3.20}$$

Si resolvemos el problema de autovalores generalizado:

[3]Es decir, que los vectores del espacio n dimensional original son combinación lineal de m vectores base, siendo el resto de variabilidad atribuible a ruido.

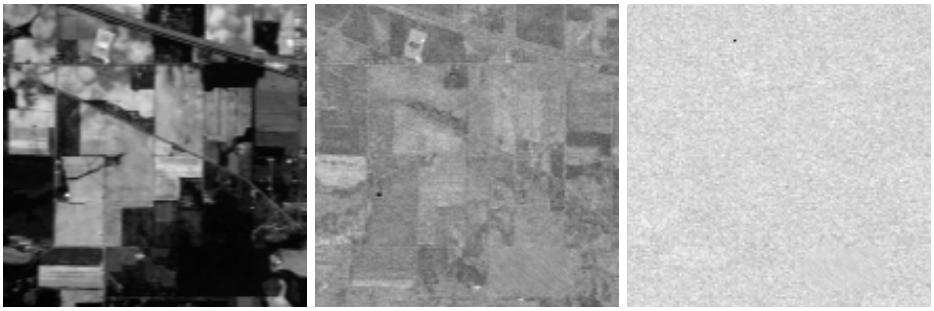


Figura 3.6: En orden, la primera, quinta y vigésima componente de la transformada MNF de una imagen. Es claro que la cantidad de ruido es menor para las primeras componentes.

$$S_{\mathcal{N}}\boldsymbol{w} = \lambda S\boldsymbol{w} \tag{3.21}$$

Tenemos que $W = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n] \in \mathcal{M}_{n\times n}$

El conjunto T de datos transformados con MNF se puede expresar [40] entonces como $T = W^T Z$, donde Z es el conjunto X centrado en 0 restando la media $\bar{\boldsymbol{x}}$ (Ecuación (3.14)).

En este caso $T \in \mathcal{M}_{n\times p}$. En cada t_i, conforme $i \to n$, la componente tiene cada vez más ruido, como puede verse en la Figura 3.6. Podemos quitar las $m - n$ componentes más ruidosas para reducir la dimensión de T.

3.6.4. Análisis de componentes extremas (VCA)

El análisis de componentes extremas (Vector Component Analysis (VCA)) fue diseñado [41] como algoritmo de desmezclado espectral: Partiendo de una imagen hiperespectral es capaz de extraer las firmas espectrales de los materiales de la imagen, y dar cada píxel como combinación lineal de las mismas.

VCA parte de la descomposición en valores singulares (sección 3.6.1.1) o del análisis de componentes principales (sección 3.6.1), y los refina para conseguir mejores resultados. [4]. Define además la matriz Y, resultante de dividir cada muestra de X entre su proyección sobre la media $\bar{\boldsymbol{x}}$.

Para ello va generando progresivamente un subespacio definido por vectores de una matriz A, que comienza con $\boldsymbol{e}_1 = [0, \ldots, 0, 1]^T$ como único vector del subespacio. Tras esta inicialización, se itera:

- Se genera un vector aleatorio u.
- Se calcula el vector $\boldsymbol{f}$:

$$\boldsymbol{f} = \frac{\left(I - AA^{\#}\right)\boldsymbol{u}}{||\left(I - AA^{\#}\right)\boldsymbol{u}||} \tag{3.22}$$

  Donde $A^{\#}$ es la pseudoinversa de A. Así se asegura la ortonormalidad de $\boldsymbol{f}$ respecto al subespacio generado por A.

- Se calcula la proyección de Y sobre $\boldsymbol{f}$, y se actualiza la columna iésima de A con la muestra de Y que maximiza dicha proyección. El índice I_i que maximizaba la proyección se guarda.

[4] Nos centramos aquí en la versión con SVD, debido a las dificultades a la hora de repetir los resultados propuestos en el artículo [41] con PCA

Tras repetir este proceso m veces, se toman los puntos de X de índices $I_i, i : 1, \ldots, m$ como vectores $\boldsymbol{w}_i$, y se forma la matriz de transformación $W = [w_1, \ldots, w_m]$. La pseudoinversa de W se utiliza para reconstruir los datos originales. Por tanto $f(X) = XW$ y $g(T) = pinv(W)T$.

3.6.5. PCA con cuantización vectorial (VQPCA)

Hasta ahora, todos los métodos vistos hacen reducciones *lineales* de los datos. Crean cierta matriz de proyección con la que transformar $X \to T$, y otra de recuperación para aproximar $T \to X$.

El análisis de componentes principales con cuantización vectorial (VQPCA) divide mediante cuantización vectorial los datos de entrada en grupos. Posteriormente aplica PCA por separado a cada uno, consiguiendo una reducción lineal *a trozos*. Así se mejoran las técnicas lineales, consiguiendo rendimiento similar o superior incluso a técnicas no lineales [42].

La cuantización vectorial subdivide un conjunto de vectores X en c conjuntos.

VQPCA recibe como entrada los datos X, así como el número c de conjuntos en los que dividir antes de aplicar PCA. En primer lugar son generados los índices $C = \{c_1, \ldots, c_p\}$ que indican a qué conjunto X_j pertenece cada muestra:

$$X_j = \{x_i \shortmid\shortmid c_i = j\}, \quad j \in \{1, \ldots, c\} \tag{3.23}$$

Posteriormente se calcula PCA para cada X_j por separado, y se generan los T_j correspondientes, así como las funciones de recuperación $g_j : T_j \to X_j$.

Para poder recuperar X, necesitamos conservar:

- C, para poder separar T en sus subconjuntos, puesto que $T_j = \{t_i \shortmid\shortmid c_i = j\}$.
- $T = \cup_{j=1}^{c} T_j$, que será separado con C.
- $g_j, j \in \{1, \ldots, c\}$, para poder recuperar cada conjunto con su PCA correspondiente.

Guardar T es inevitable para cualquier reducción de dimensionalidad, pero VQPCA necesita considerablemente más cosas que los demás métodos. C, sin ir más lejos, supone un índice por muestra, y las funciones g_j son matrices grandes de las cuales habrá que guardar bastantes si c es grande.

Tanto T como $g_j, j \in \{1, \ldots, c\}$ son demasiado aleatorios como para ser comprimidos, pero C presenta redundancia espacial, ya que por lo general muestras cercanas tenderán a pertenecer a los mismos X_j. Por tanto podemos añadir compresión aritmética adaptativa, que reducirá bastante su tamaño total.

Podemos hacernos una idea de las ventajas de VQPCA viendo la Figura 3.7. En ella observamos como al hacer PCA dentro de cada grupo generado con VQ, conseguimos mejor separación que con PCA tradicional sobre la totalidad de los datos.

3.6.6. Autocodificadores

Los autocodificadores aprovechan las redes neuronales para comprimir información. Se forma una red con varias capas, donde tanto la primera como la última son del mismo tamaño. La red se entrena para que, ante una entrada determinada, produzca en la salida exactamente el mismo resultado.

Se alimenta un vector $\boldsymbol{x} = \{x_1, \ldots, x_n\}$ a la red, que tiene n neuronas de entrada, y se obtiene otro vector $\tilde{\boldsymbol{x}} = \{\tilde{x}_1, \ldots, \tilde{x}_n\}$ en las n neuronas de salida, que se espera sea igual que el primero, o lo aproxime lo suficiente.

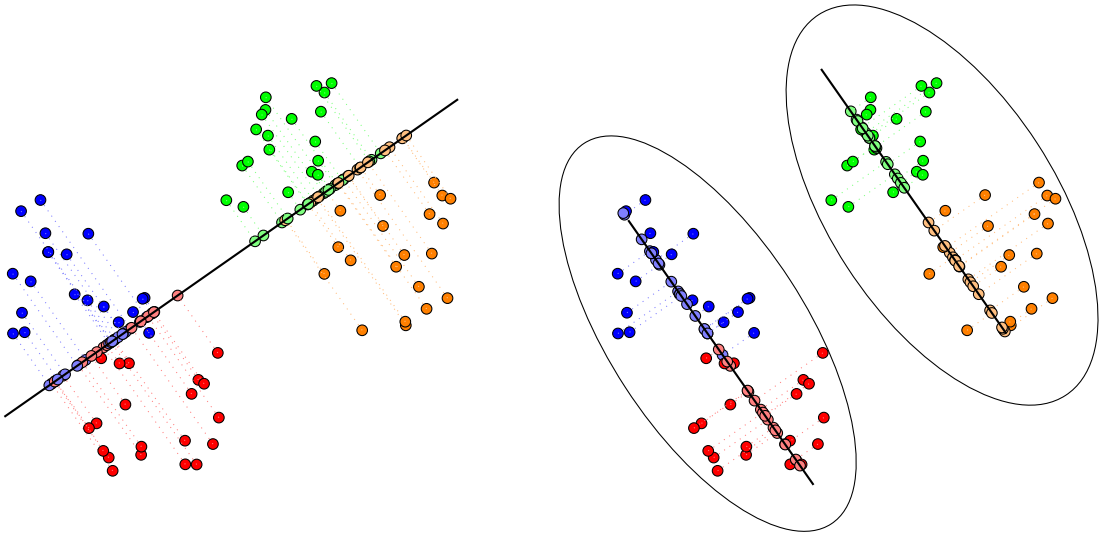


Figura 3.7: A la izquierda, el resultado de aplicar PCA a un conjunto de puntos. Vemos como puntos diferenciados se proyectan en la misma zona. A la derecha, VQPCA. La separación en grupos hace que los PCA posteriores consigan mejor resultado restringidos a su grupo.

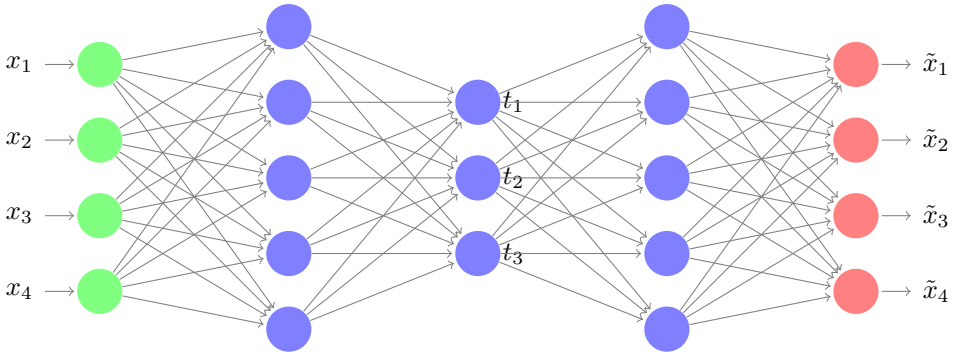


Figura 3.8: Observamos que la capa de entrada y de salida son de la misma dimensión. En una de las capas ocultas la dimensión es menor, y por tanto se puede utilizar para realizar compresión.

El truco para la compresión está en las capas ocultas, de las cuales una al menos consta de un número $m < n$ de neuronas. Una vez creada la red completa, bastará guardar para cada entrada $\boldsymbol{x}$ los valores de las salidas ($\boldsymbol{t}$) de dicha capa oculta, así como las capas siguientes necesarias para la reconstrucción.

El principal inconveniente de este método es el tiempo requerido para entrenar la red. Tenemos cientos de miles de entradas, y queremos entrenar con todas para tener una buena reconstrucción del conjunto[5]. Al ser similares entre ellas, es posible hacer un muestreo aleatorio y entrenar con solo un subconjunto sin deteriorar mucho el comportamiento. Aun así, los tiempos de entrenamiento van desde varios minutos hasta varias horas.

Una potencial ventaja es la posibilidad de reutilizar la red, si no ha sido sobreentrenada, si los datos a comprimir son similares para los diferentes conjuntos que comprimamos. En cualquier caso, la reutilización del reductor dimensional también sería posible como métodos como PCA o VCA.

[5]De hecho, queremos sobreentrenar la red para que funcione lo mejor posible con el conjunto de entrenamiento

Capítulo 4

Del celuloide a JPEG2000

La química fue la reina de la fotografía desde su invención en la primera mitad del siglo XIX. Materiales fotosensibles cambiaban al exponerse a la luz y conseguían capturar recuerdos que hasta entonces sólo podían retratarse con brocha fina.

El proceso de captura y revelado, si bien fue mejorando con los años, siempre mantuvo su esencia. Se tomaba un negativo en la cámara exponiendo la película a la luz. Para evitar su deterioro, se fijaba la imagen químicamente eliminando las propiedades fotosensibles en la oscuridad (Figura 4.2). Posteriormente el negativo era utilizado para obtener copias de la imagen, que de nuevo mediante materiales fotosensibles volvían a invertirse para dar lugar a copias positivas que poder guardar y disfrutar.

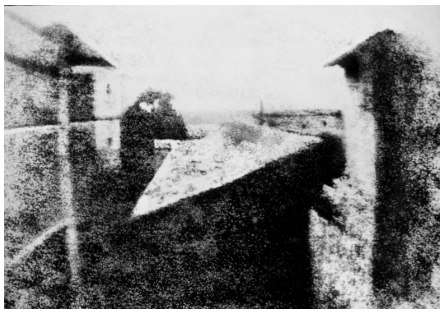


Figura 4.1: Versión retocada de la fotografía más antigua de la que se tiene copia (c. 1825), tomada por Joseph Nicéphore Niépce. (Dominio público)

Este proceso analógico limitaba la resolución al tamaño de las partículas fotosensibles, y el espacio que ocupaban las imágenes al tamaño del papel positivo. Si bien hoy en día es habitual tener varios álbumes de recuerdos en casa, la cantidad de fotografías almacenadas en ellos es irrisoria si la comparamos con la cantidad de imágenes que puede llegar a enviar nuestro cuñado favorito por *Whatsapp*.

En 1975 [44] cambió el mundo de la fotografía de la mano de Steven Sasson. La primera cámara digital eliminaba la principal pega de las analógicas, permitiendo reutilizar el sensor

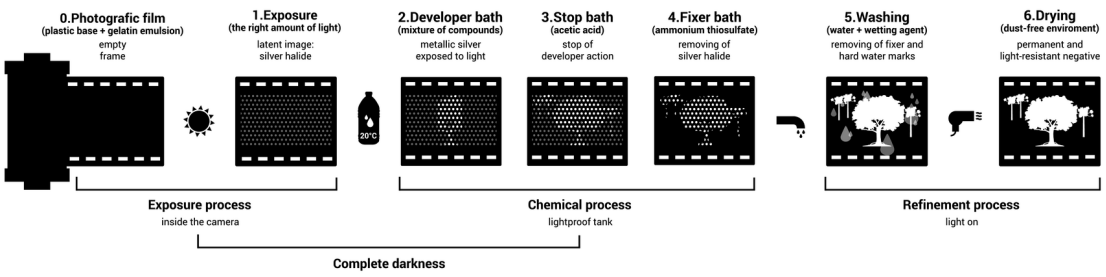


Figura 4.2: Proceso de revelado químico. El material fotosensible se expone a la luz, y mediante una serie de procesos se fija la imagen. [43].

(hasta entonces película) para tantas imágenes como quisiéramos. Con una resolución de 0.01 megapíxeles, la primera cámara no generaba gran cantidad de información. Con el paso de los años salen al mercado las primeras DSLR[1] [45] (Figura 4.3, que poco a poco se acercaban a la barrera del megapíxel.

El espacio necesario para almacenar las fotografías aumentaba, y el hecho de tener un sensor reutilizable hacía que cada vez se capturasen más instantáneas. Llegó un punto en que fue necesario comenzar a comprimir los datos fotográficos para reducir el tamaño de las pilas de disquetes. Desde 1986 hasta 1992 [21] el Joint Photographic Experts Group (JPEG) desarrolló el que se convertiría en el estándar de compresión más utilizado hasta el día de hoy, y que en un alarde de imaginación llamaron estándar JPEG [47].

Figura 4.3: Una de las primeras cámaras DSLR [46].

4.1. JPEG

JPEG es un estándar de compresión con pérdida. Su objetivo es reducir significativamente el tamaño de las imágenes con un algoritmo de baja complejidad, introduciendo la menor distorsión posible. Su adopción fue rápida, e incluso a día de hoy sigue utilizándose como compresión principal en las cámaras digitales, e incluso recibe mejoras [48] de la mano de grandes empresas.

El primer paso en JPEG es separar la información sobre la imagen, habitualmente representada por canales de color Red Green Blue (RGB), en canales de brillo y color YCbCr (ver Figura 4.4). El ojo humano es mucho más sensible a diferencias en intensidad de luz que a cambios de color, y esta separación permitirá una compresión más agresiva sobre los canales Cb y Cr, que contienen la información del color.

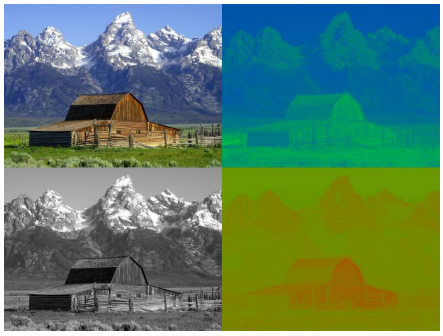


Figura 4.4: Separación [49] en los canales Y (brillo), Cb (azul) y Cr (rojo).

Posteriormente se divide la imagen en pequeñas baldosas (o bloques) cuadradas de 8×8 píxeles. Cada baldosa (tanto si es de brillo como de color) se comprimirá independientemente. Esto hace que la carga computacional del algoritmo sea mucho menor, al no necesitar operar sobre la imagen entera de golpe.

La primera compresión llevada a cabo es una reducción de la precisión en las baldosas de color. Si habitualmente se utilizan $8bits$ por muestra, se verán reducidos a 4 o incluso 2 en las baldosas Cb y Cr. La componente Y suele dejarse intacta.

A continuación se aplica otro nivel de compresión a cada baldosa mediante la Transformada Discreta del Coseno (DCT) (Figura 4.5). Esta transformación genera un nuevo bloque de 8×8 a partir de los coeficientes (valores) del original, con la particularidad de que los coeficientes cercanos a la esquina inferior derecha suelen ser de valor bajo.

Además, debido a las características de la transformación y del ojo humano, la información representada por esos coeficientes de la esquina inferior derecha es poco importante. Para reducir

[1] Digital Single-Lens Reflex, o reflex digital de lente única.

la información a codificar, cada coeficiente se divide entre un factor de ajuste, que es mayor cuanto más cerca de dicha esquina estamos. Con ello se consigue reducir información poco importante, mientras que mantenemos el grueso de la imagen intacto.

$$\begin{bmatrix} 52 & 66 & 70 & 73 \\ 63 & 122 & 154 & 69 \\ 67 & 104 & 126 & 70 \\ 87 & 68 & 65 & 94 \end{bmatrix} \overset{\text{centrado}}{\rightarrow} \begin{bmatrix} -76 & -62 & -58 & -55 \\ -65 & -6 & 26 & -59 \\ -61 & -24 & -2 & -58 \\ -41 & -60 & -63 & -34 \end{bmatrix} \overset{\text{dct}}{\rightarrow}$$

$$\overset{\text{dct}}{\rightarrow} \begin{bmatrix} -415 & 27 & 56 & 0 \\ -49 & -15 & -10 & 2 \\ 12 & -4 & -2 & 3 \\ 0 & -4 & -1 & 2 \end{bmatrix} \overset{\text{cuantizacion}}{\rightarrow} \begin{bmatrix} -26 & 2 & 2 & 0 \\ -4 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \overset{\text{zig-zag}}{\rightarrow}$$

$$\overset{\text{zig-zag}}{\rightarrow} \{-26, -4, 2, 2, -1, 1, 0, \ldots\} \overset{\text{codificación}}{\rightarrow} \texttt{010101100} \quad (4.1)$$

Tras este proceso disponemos de una baldosa con muchos ceros en la esquina inferior derecha. Esta característica hace que se pueda aplicar una codificación entrópica (sección 3.4) siguiendo un recorrido en zig-zag de la baldosa, comprimiendo aún más los datos que ya habían sido reducidos en magnitud. Podemos ver el flujo del proceso en bloques de 4×4 en la Ecuación (4.1).

Este proceso es reversible, y por tanto invirtiéndolo volvemos a tener las baldosas originales, que uniéndose forman la imagen inicial. Eso sí, hemos ido perdiendo cierta información en cada paso (dependiendo de lo que reduzcamos los canales del color, de lo que dividamos los coeficientes de la DCT...) y por tanto recuperar el original no será posible, sino solo una aproximación, más cercana al original cuanta menos compresión hayamos aplicado.

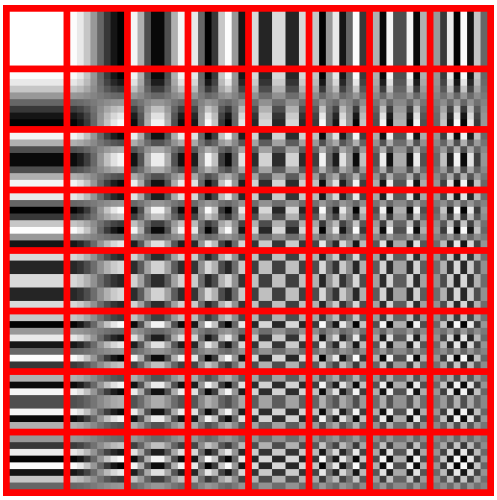


Figura 4.5: La DCT representa cada bloque de 8×8 como combinación lineal de los 64 bloques de esta figura [50].

Cuando se diseñó JPEG, la potencia de los procesadores estaba muy lejos de lo que existe hoy en día. Era necesario por tanto utilizar técnicas "baratas" en cuanto a coste computacional, como la división en baldosas pequeñas. Con el tiempo y con la mejora de los procesadores, nuevas técnicas mucho más potentes, y sin las desventajas de JPEG fueron surgiendo. Entre ellas el siguiente estándar de JPEG: JPEG2000.

4.2. JPEG2000

JPEG2000 nació como sucesor de JPEG, mejorando las debilidades de JPEG, como los artefactos debidos a la división en bloques (Figura 4.6), y añadiendo nuevas funcionalidades para permitir una compresión más flexible (definición de zonas de interés que se conservan con más calidad, integración de nuevos metadatos, diferentes modos de compresión opcionales...).

Si JPEG utilizaba una compresión *local* (a nivel de baldosa), JPEG2000 se centra en una compresión *global*. De manera similar a como la DCT concentraba la información importante en una esquina de cada baldosa, JPEG2000 emplea una transformación de ondícula (sección 4.2.2)

para hacer algo similar a nivel global: La esquina superior izquierda de la transformación contendrá una versión reducida de la imagen, mientras que el resto contendrán información extra para ir completando y definiendo cada vez más dicha versión reducida. Estas transformaciones se aplican también a nivel de canal de color, aprovechando las características del ojo con el espacio de color YCbCr.

Posteriormente se parte la imagen en bloques y se codifica cada uno por separado. Los bloques son de tamaño bastante superior a los de JPEG (hasta 4096 muestras frente 64 de JPEG), lo cual permite una mayor eficiencia, sin elevar excesivamente los requerimientos de memoria.

Tanto la transformación de ondícula como la codificación en bloques se hacen de manera *progresiva*. Esto implica que a la hora de decodificar, se obtiene un resultado aproximado rápidamente, el cual se va refinando conforme se recibe más información. Esto permite "cortar" el flujo de datos en cualquier momento, obteniendo una imagen comprimida del tamaño deseado. También permite fijar una calidad deseada, y transferir los datos justos para que se alcance la misma.

Pero pese a estas ventajas, el sobrecoste de JPEG2000 frente a JPEG y la complejidad de su algoritmo ha hecho de su adopción algo meramente anecdótico. Aunque consigue una calidad muy superior a la de JPEG, JPEG es *suficientemente* bueno, y la complejidad añadida no compensa en muchos casos el cambio. Aun así, ha encontrado su nicho en la fotografía profesional y aplicaciones científicas, y sus técnicas son muy utilizadas para la compresión de vídeo o imágenes en internet, donde ahorrar ancho de banda siempre es deseado.

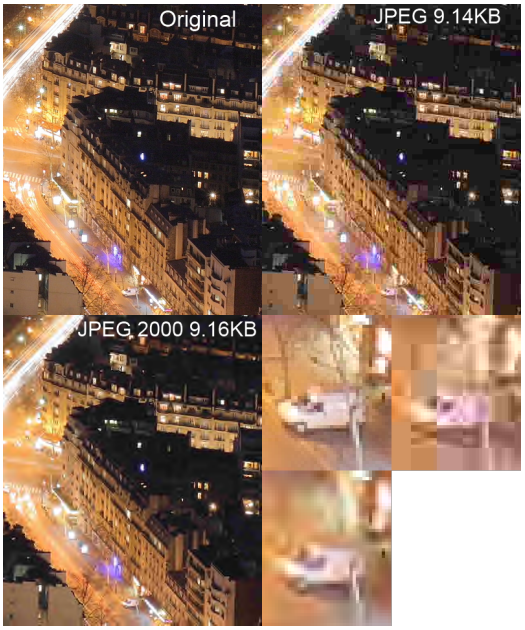


Figura 4.6: En esta imagen [51] apreciamos la diferencia básica entre JPEG y JPEG2000. JPEG genera bloques de 8×8 en compresiones agresivas, mientras que JPEG2000 suaviza en conjunto la imagen original.

4.2.1. Cambio de espacio de color

Una imagen habitualmente consta de tres canales de color: rojo, verde y azul (RGB). JPEG ya utilizaba la transformación RGB→YCbCr, que recordemos separaba la componente de brillo de las de color. JPEG2000, además de permitir utilizar esta transformación (tanto en modalidades de compresión con pérdida como sin pérdida) es capaz de realizar transformaciones sobre imágenes con más de tres canales:

- Transformaciones lineales mediante multiplicación por matrices.
- Transformaciones mediante modelos predictivos, que utilizan un entorno de cada píxel para realizar la predicción.
- Transformaciones de ondícula sobre los canales de cada píxel.

Tras este cambio de color, se puede rebajar la cantidad de bits utilizada en cada canal. Aplicando la misma idea que en JPEG, los canales Cb y Cr serían claros candidatos para ello, pues el ojo los percibe peor y no importa perder algo de información. Esta operación induce, por supuesto, una pérdida en la calidad final de la imagen.

En cualquier caso, suele ser mejor, en términos de calidad, realizar este rebajado tras la transformación de ondícula, en el paso de cuantización que veremos con posterioridad.

4.2.2. Transformaciones de ondícula

Tras un cambio en el espacio de colores, el siguiente paso en JPEG2000 es aplicar una transformación de ondícula. Las transformaciones de ondícula van a transformar una matriz o imagen en otra de las mismas dimensiones, y por tanto se aplican a nivel de canal (**banda**) en el espacio de colores transformado.

El objetivo es concentrar la "energía" de la banda, o las características más importantes en una porción de la matriz transformada (ver Figura 4.8). El resto de los datos transformados contendrán poca "energía" o información, y por tanto se podrán comprimir con mayor facilidad.

Las transformaciones de ondícula se definen sobre vectores unidimensionales, existiendo después extensiones a dos dimensiones. El vector de entrada se toma como una señal muestreada en intervalos regulares, y se le aplican dos filtros: uno paso bajo y otro paso alto, ambos en forma de kernels.

Definición 3 *Un **kernel** es una función $k : \mathbb{R}^n \to \mathbb{R}^n$, que toma cada elemento del vector de entrada junto con los adyacentes y los combina para obtener el elemento del mismo índice en el vector de salida. Los kernels se suelen representar por matrices, por ejemplo:*

$$K = \begin{bmatrix} -\frac{1}{2} & 1 & -\frac{1}{2} \end{bmatrix}$$

Es un kernel $K : v = \{v_1, \ldots, v_n\} \to w = \{w_1, \ldots, w_n\}$ donde:

$$w_i = -\frac{1}{2}v_{i-1} + v_i - \frac{1}{2}v_{i+1}$$

En los extremos esta definición puede causar problemas por no existir las muestras referenciadas. Una solución es ignorarlas, y hacer que el vector de salida sea más pequeño. Habitualmente se toma el vector como circular para poder tener muestras, o se hace la imagen espejo del vector en los bordes para tener muestras del interior.

- El filtro paso bajo mantiene intactas las características suaves del vector. Genera otro vector similar al original, donde los picos y valles se han suavizado.
- El paso alto elimina estas características sostenidas en el tiempo, y deja pasar las frecuencias o variaciones rápidas en la señal, generando un nuevo vector donde se almacenan las características más "ruidosas".

A partir de estos vectores transformados es posible recuperar la señal original, incluso si rebajamos a la mitad su longitud. Con esto conseguimos una transformación con el mismo número de muestras de entrada que de salida con la ventaja de concentrar la información de la señal en la mitad de paso bajo (ver Figura 4.7). Debido a esto, estas transformaciones funcionan especialmente bien cuando la señal tiene cierta continuidad.

Definición 4 *Dada una sucesión $S = \{a_i\}_{i=1}^n$, una **transformación de ondícula** es una función invertible $W : \mathbb{R}^n \to \mathbb{R}^n$.*

W se aplica utilizando una tupla de kernels (K_l, K_h) que, aplicados sobre S actúan respectivamente como filtros paso bajo y alto. $W(S) = (K_l(S), K_h(S)) = (L, H)$ produce sendas sucesiones $L = \{l_i\}_{i=1}^n$ y $H = \{h_i\}_{i=1}^n$. Ambas sucesiones se rebajan a la mitad de muestras:

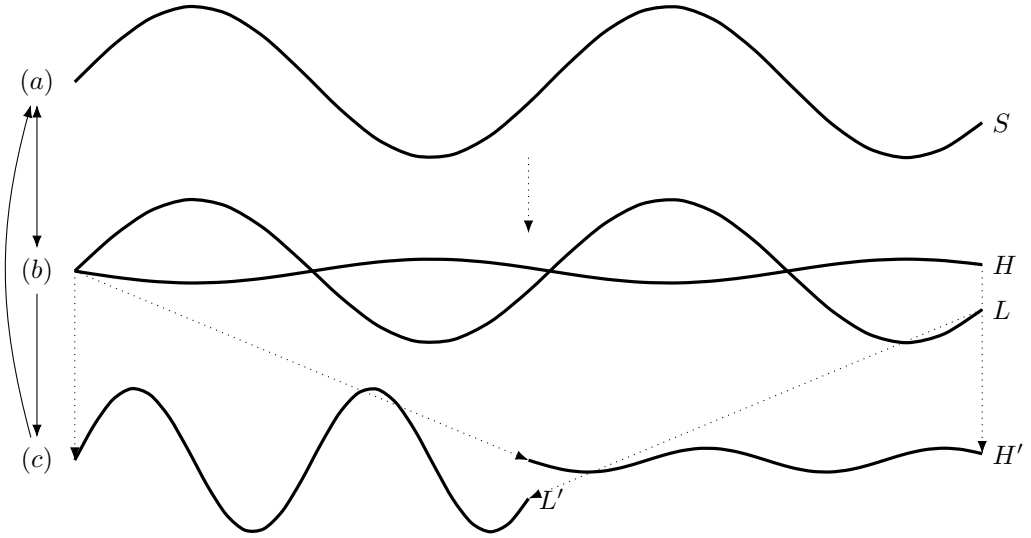


Figura 4.7: (a) es la serie original, (b) muestra las sucesiones resultantes de los filtros paso bajo y alto, (c) resulta de tomar la mitad de muestras de (b).

$$L' = \{l_i\}_{i\in 2\mathbb{N}} \quad H' = \{h_i\}_{i\in 2\mathbb{N}} \tag{4.2}$$

Y por último se entrelazan para formar $S' = \{a'_i\}_{i=1}^{n}$*:*

$$a'_i = \begin{cases} l_{i/2} & i \in 2\mathbb{N} \\ h_{(i-1)/2} & c.c. \end{cases} \tag{4.3}$$

Para la inversión, se genera otra tupla de kernels (K'_l, K'_h)*, y siguiendo un proceso análogo se obtiene* S *de vuelta partiendo de* S'*. La cadena de transformaciones se puede ver en la Figura 4.7.*

Observando (c) en la Figura 4.7, vemos que, si bien la primera parte es similar a la serie original, el segundo tramo (paso alto) tiene los valores muy sesgados hacia cero, y esta redundancia será aprovechable para una buena compresión. Y no solo eso, las transformaciones de ondícula se pueden aplicar recursivamente sobre L para seguir sesgando los datos.

Al tratar con imágenes, ya no disponemos de una serie de manera tan clara. Por fortuna, se pueden extender las ondículas a matrices bidimensionales, con tan solo aplicarlas primero en una dirección (por ejemplo verticalmente) y después en la otra (horizontalmente). Así, conseguimos resultados como los observados en la Figura 4.8. La zona que tan solo ha pasado por filtros de paso bajo se asemeja a una versión reducida de la imagen original, mientras que el resto de la imagen es ahora mucho más uniforme y, por tanto, comprimible.

Definición 5 *En dos dimensiones, los sectores resultantes de aplicar la transformación de ondícula (conocidos como* ***subbandas****) siguen la siguiente nomenclatura:*

LL *Paso bajo horizontal y paso bajo vertical. Contiene la mayor parte de información sobre la imagen y es difícil de comprimir.*

LH *Paso bajo horizontal y paso alto vertical. La redundancia vertical es eliminada, y presenta redundancia horizontal.*

HL *Paso alto horizontal y paso bajo vertical. Al contrario que LH, la redundancia aquí es vertical.*

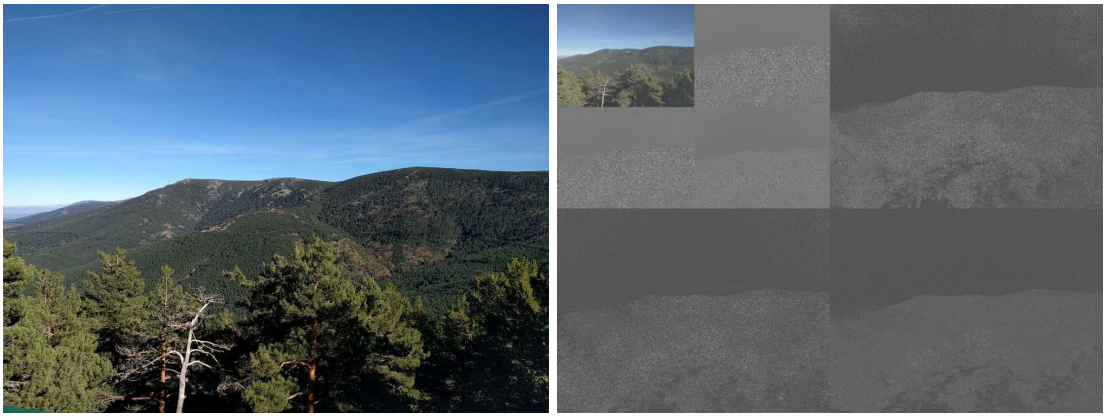


Figura 4.8: A la izquierda, la imagen original. A la derecha, la imagen a la cual se ha aplicado transformación de ondícula dos veces. Se observa como en la imagen procesada, la información a guardar se concentra en la esquina superior izquierda.

HH *Paso alto horizontal y paso alto vertical. La redundancia se elimina en ambas direcciones, si bien aparecen artefactos diagonales.*

Cuando se aplica varias veces la transformación (siempre sobre LL), las subbandas resultantes se denotan con un subíndice (p.ej: LH_2) que indica en qué iteración de la transformación de ondícula se ha creado.

Existen numerosos tipos de transformaciones de este tipo, y todos ellos, teóricamente, son capaces de reconstruir la señal original a partir de la transformada, gracias a la invertibilidad de las funciones W. En la práctica, la utilización de números con precisión decimal finita hace que en ocasiones no sea del todo cierto, y debido a redondeos se puedan introducir errores. Como solución, existen las transformaciones de ondícula *sin pérdida*, que aseguran que la señal original puede ser recuperada, si bien los resultados que ofrecen de cara a la compresión empeoran ligeramente, pues sus filtros paso baja y alta separan peor la señal.

4.2.3. Cuantización

La transformación de ondícula se realiza sobre números en coma flotante (exceptuando la versión sin pérdida, que trata con enteros). Tratar con coma flotante implica cierto grado de imprecisión, que hará que la imagen comprimida tenga algo de distorsión. En este momento es cuando se elige la cantidad de distorsión que va a existir.

La codificación, que es realizada posteriormente, necesita números enteros, en formato signo-magnitud. Es necesario redondear los resultados de la transformación de ondícula de alguna manera, para obtener datos utilizables por el codificador.

La solución propuesta para JPEG2000 es el conocido como *cuantizador uniforme con zona muerta*. El cuantizador opera independientemente sobre cada banda de la imagen siguiendo la fórmula:

$$q_b[n] = sign(y_b[n]) \left\lfloor \frac{|y_b[n]|}{\Delta_b} \right\rfloor \tag{4.4}$$

Donde la notación en la fórmula es la siguiente:

n Posición dentro de una subbanda. n es una abstracción para una posición bidimensional, ya que las muestras son elementos de matrices.

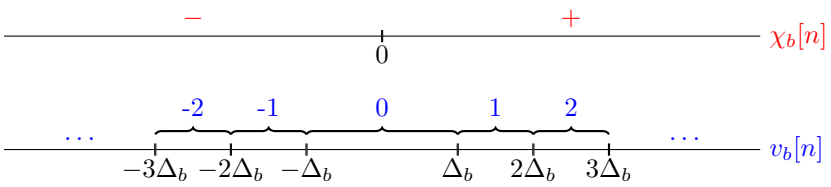


Figura 4.9: Observamos en la imagen cómo se realiza la cuantización. Por un lado, el signo de la muestra original es conservado. Por otro, Δ_b determina la magnitud asociada. Cerca del cero, el intervalo de redondeo es el doble que los demás, esta es la "zona muerta" de donde toma el nombre el cuantizador.

$y_b[n]$ Muestra de la subbanda b en la posición n, normalizada al intervalo $(-1/2, 1/2)$ tras la transformación de ondícula.

Δ_b Paso para la cuantización de la subbanda b.

Como las muestras cuantizadas han de expresarse en formato signo-magnitud, podemos definir los valores $\chi_b[n] \in \{-1, 1\}$ y $v_b[n]$ como signo y magnitud de los valores cuantizados:

$$\chi_b[n] = sign(y_b[n]), \quad v_b[n] = \left\lfloor \frac{|y_b[n]|}{\Delta_b} \right\rfloor \tag{4.5}$$

Cuanto menor sea el valor Δ_b, más posibles valores podrá tomar $v_b[n]$. Tendremos más precisión en la reconstrucción posterior de los valores originales, y el número de bits necesarios para almacenar $v_b[n]$ crecerá.

4.2.4. Codificación en bloques

Si bien la transformación de ondícula es global, la característica local de los kernels permite hacerla poco a poco, sin necesidad de cargar la imagen completa en memoria. El siguiente paso es la codificación de los coeficientes resultantes de la transformación, y también debemos evitar una sobrecarga en memoria. La solución que acordó el comité de JPEG2000 fue la de realizar la codificación por bloques de tamaño reducido[2].

Es interesante que los bloques se puedan decodificar progresivamente mejorando la calidad de la imagen reconstruida poco a oco, por lo que su codificación se hace de la misma manera: Cada bloque se separa en planos de bits, y se codifican de más a menos significativos. No solo eso, cada plano de bits se codifica hasta en tres pasadas diferentes, en las cuales se irá codificando la información que se *prevé* más importante, mediante modelos predictivos.

Introducimos a continuación nomenclatura, similar a la utilizada en cuantización, para referirnos a los elementos de un bloque:

j Posición dentro de un bloque. j es una abstracción para una posición bidimensional, ya que las muestras son elementos de matrices.

$y[j]$ Muestra en la posición j.

$v^{(p)}[j]$ Bit p-ésimo de la muestra $y[j]$.

$\chi[j]$ Signo de la muestra $y[j]$.

[2]Los bloques contienen hasta 4096 muestras cada uno, con un tamaño máximo de 1024 muestras de lado. La forma habitualmente empleada son cuadrados de 64×64.

Y ahora, ¿cómo codificamos las muestras $y[j]$?. Hablábamos antes de un modelo predictivo que nos iba a ayudar a hacer una codificación más eficiente. Para hacerlo funcionar, vamos a definir un estado, la significancia, para cada muestra:

Definición 6 *La* ***significancia*** *de una muestra* $y[t]$ *se denota por* $\sigma[t]$*, y puede tomar los valores* insignificante*,* significante positiva *y* significante negativa*.*

Mientras todos los bits $v^{(p)}[j]$ *codificados de una muestra sean* **0***, la muestra será insignificante. En el momento en el que se codifique un* **1***, pasará a ser significante positiva o significante negativa, según* $\chi[j]$*.*

Como los bits se codifican de más a menos significativo, la significancia de una muestra en un instante determinado nos indica si aporta algo interesante al flujo comprimido hasta entonces.

Es precisamente este concepto de significancia en base al cual definimos las tres pasadas necesarias para codificar cada plano de bits de un bloque, pues incluso dentro del mismo plano, algunos valores tendrán más posibilidades de ser interesantes que otros. Estas probabilidades sesgadas son las que, codificadas por separado, mejoran la tasa de compresión y la calidad de una descompresión progresiva:

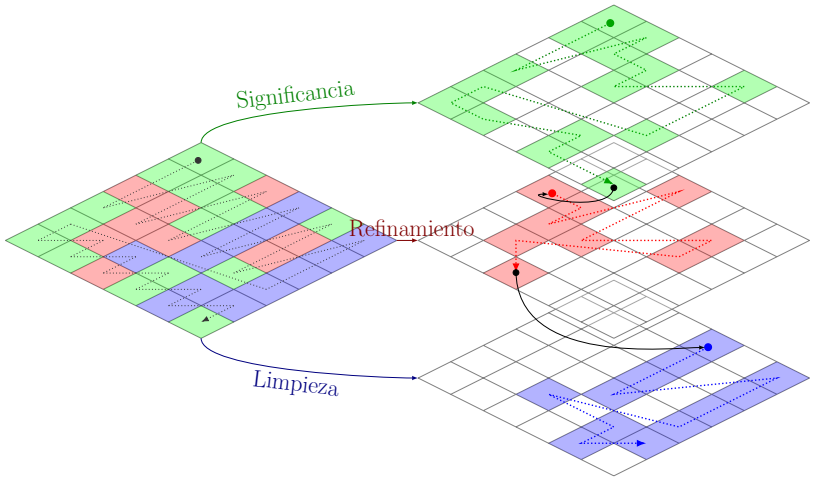


Figura 4.10: Cada plano de bits (simplificado en este caso a 6×6) es codificado en tres pasadas. El recorrido en zig-zag sigue la tira de altura 4, y continúa en tiras de menor altura si el lado del bloque no es múltiplo de 4.

Propagación de significancia: Se codifican las muestras insignificantes que se cree que van a cambiar a significantes. Por ejemplo, cuando una muestra es insignificante y muchas de sus vecinas son significantes, parece razonable pensar que la probabilidad de que cambie su significancia es alta.

Refinamiento: Cuando una muestra ya es significante, el comportamiento de los bits que quedan por codificar no es tan predecible. Estos bits se codifican en esta pasada especial. Mantenemos además un indicador por muestra que activamos tras codificar el primer bit de refinamiento, con el fin de mejorar nuestro modelo predictivo.

Limpieza: Todo lo que no se codifica en las anteriores pasadas se codifica aquí. Por lo general serán zonas del plano sin importancia, y con alta probabilidad encontraremos largas cadenas de ceros. Para tratarlas, esta pasada está equipada con un codificador de carrera, que codifica en poco espacio las cadenas uniformes.

Hay una excepción a las pasadas de codificación, y es el plano de signo. Los coeficientes a codificar se encuentran en formato signo-magnitud. Todos los planos de magnitud se codifican de la manera mencionada, y el signo únicamente es codificado cuando una muestra pasa a ser significante. Por tanto la codificación del plano de signo se intercala con todas las pasadas por los demás planos.

En los planos de magnitud, el recorrido para la codificación hará una especie de zig-zag por el bloque: Las filas se agrupan de 4 en 4 en tiras, y cada tira se recorre por columnas (Figura 4.11). Este recorrido hace que los bits que se visitan seguidos presenten características similares. Para no codificar un bit en varias pasadas, se van marcando según son codificados. Si un bit ha sido ya codificado en una pasada anterior, el recorrido en zig-zag se lo salta.

Los parecidos entre bits cercanos inducidos por el orden de recorrido son los que hacen que la codificación se pueda mejorar con modelos predictivos. Aunque es más evidente su uso en la pasada de propagación de significancia, también es útil en las otras dos. Existe un modelo para cada contexto.

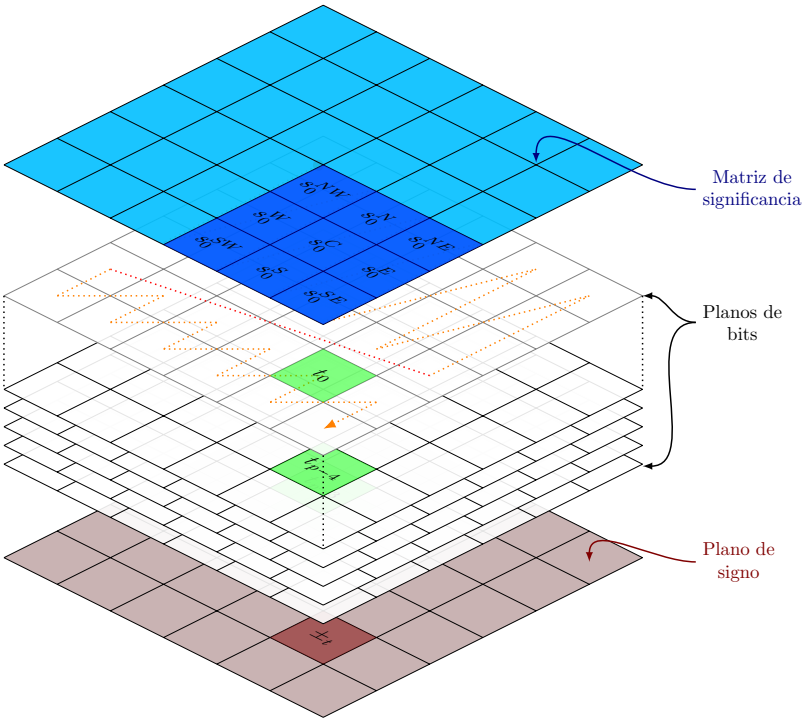


Figura 4.11: Observamos la codificación en un bloque reducido de 6×6. El recorrido en zig-zag sigue la tira de altura 4, y continúa en tiras de menor altura si el lado del bloque no es múltiplo de 4. Los planos de bits se codifican de más (0) significativo a menos (p) (asumiendo una profundidad de $p+1$ bits). El plano de signo se codifica según necesidad. Todo ello utilizando los contextos proporcionados por la matriz de estado, donde se consultan bloques de 3×3 vecinos.

Definición 7 *Un **contexto** representa un patrón en los datos a codificar. Los contextos se generan [29] partiendo del estado de significancia de las muestras que rodean a la que estamos codificando, en un cuadrado de 3×3 (Figura 4.11). También se tiene en cuenta la subbanda (LL, LH, HL, HH) a la que pertenece la muestra. La cantidad de combinaciones de estados posibles es muy alta, y para simplificar se colapsan todas a 16 contextos distintos.*

Los contextos son generados de manera diferente en cada una de las pasadas, contando con una generación especial en el caso del contexto del bit de signo. Como ejemplo, se muestran en la Figura 4.12 varios vecindarios que dan lugar al mismo contexto.

Figura 4.12: Ejemplo de generación del contexto. En rojo muestras significativas, gris indica insignificancia. Todos los vecindarios aquí mostrados generan el contexto seis en la pasada de significancia para la subbanda LH. Pueden apreciarse las simetrías. Este contexto indica cierta estructura vertical y/o diagonal en el entorno.

Tanto el contexto como el bit a codificar se envían al codificador aritmético, que será el encargado de actualizar los modelos y generar, en última instancia, la cadena de bits comprimidos.

4.2.5. Codificador aritmético MQ

El codificador de bloque, en esencia, genera pares $(bit, contexto)$, pero no realiza la codificación *per se*. Para ello, estos pares son alimentados al codificador aritmético MQ.

Como su nombre indica, estamos ante un codificador aritmético, en particular del tipo binario (sección 3.3), pues así lo es el diccionario de entrada, limitándonos a los símbolos `0` y `1`.

Como sabemos, un codificador aritmético basa su eficacia en un sesgo en los elementos de entrada. Aquí no tenemos información a priori sobre qué distribución presentan los datos, por lo que la vamos a calcular en tiempo de ejecución. Es decir, el codificador MQ es *adaptativo*.

Para cada contexto κ, mantendrá dos valores: Un símbolo $s_\kappa \in \{\texttt{0}, \texttt{1}\}$, que indica la predicción actual, y un número entero $\Sigma_\kappa \in \{0, \ldots, 46\}$, que indica el **estado** del modelo predictor de dicho contexto.

Para determinar la probabilidad del símbolo s_κ, se dispone de una tabla estática que asocia a cada estado una probabilidad fija $\bar{p} \in \{0, \ldots, 2^{16} - 1\}$. $\bar{p}$ asigna enteros al intervalo $[0, 1)$ dividiendo entre 2^{16}.

Como $\bar{p}$ es fija en cada estado, la actualización de las probabilidades se lleva a cabo mediante una tabla de transiciones, que indica el movimiento entre estados ante una predicción certera Σ_{mps} (*most probable symbol*) o fallida Σ_{lps} (*least probable symbol*).

Por último, otra tabla X_s (*switch*) nos dice, para un estado Σ_κ, si es necesario cambiar la predicción actual s_κ, pues la distribución de los bits de entrada ha cambiado su sesgo.

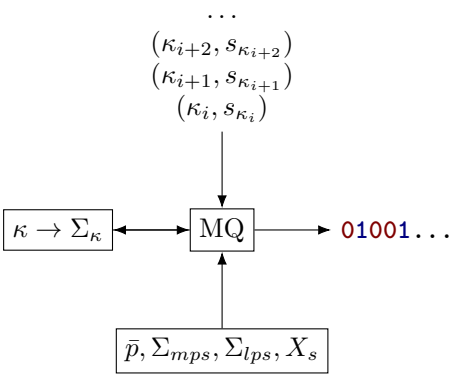


Figura 4.13: Diagrama que muestra el flujo de datos en el codificador MQ. Llegan los pares (bit, contexto). Con el contexto se extrae el estado actual, que junto a las tablas sirve para generar la salida y actualizar el estado.

Todos estos elementos forman parte de la parte predictiva (los modelos) del codificador. Recordemos de la sección 3.3 que un codificador aritmético, en esencia, generaba una fracción en el intervalo $[0, 1)$.

Este codificador mantiene únicamente una parte de la fracción activa, teniendo 16 y 28 bits respectivamente los registros A y C. Cada cierto tiempo se emiten los bits "consolidados" de la fracción que ya no van a cambiar, liberando espacio de los registros para mantenerlos en la cantidad de bits designada.

La única restricción que presenta el codificador MQ del JPEG2000 es que no puede emitir símbolos en el rango `0xff90-0xffff`, debido a estar reservados por el estándar para indicar marcadores concretos. ¡Un decodificador que encuentre una secuencia en ese rango dará por terminado su trabajo!.

Capítulo 5

Implementando el algoritmo

El gran tamaño de las imágenes hiperespectrales las hace claro objetivo de compresión. Ya sea para un manejo más sencillo, una transmisión más rápida, o un almacenamiento más eficiente, es interesante poder reducir los cientos de MegaBytes que puede llegar a ocupar una única imagen. Aquí vamos a ampliar las ideas del Capítulo 4 implementando nuestro propio algoritmo de compresión.

La gran utilidad que aporta la compresión a las imágenes hiperespectrales ha hecho que emerjan numerosas técnicas para la compresión:

- Aplicaciones banda a banda de algoritmos ya existentes en dos dimensiones [52].
- Extensión de las ideas existentes para algoritmos de compresión en dos dimensiones a tres dimensiones. Por ejemplo transformaciones de ondícula [53], o sistemas de codificación tridimensionales [54].
- Algoritmos híbridos [20] que mezclan ideas de la compresión bidimensional en las diferentes bandas (dirección espacial), y añaden una nueva perspectiva a la compresión espectral.

Las más exitosas en cuanto a nivel de compresión han sido estas últimas, y precisamente en el trabajo de [20] es donde nos basamos. Allí aplican PCA conjuntamente con una implementación comercial de JPEG2000 para realizar la compresión. Aquí se ha desarrollado un algoritmo para extender esa idea y:

- Poder aplicar no solo PCA, sino también otros algoritmos de reducción dimensional.
- Realizar una implementación propia de las partes del estándar JPEG2000 necesarias para la compresión, para tener un mayor control sobre qué y cómo se comprime. De esta manera se evitan además los sobrecostes inducidos por JPEG2000 en forma de marcadores que indican diferentes zonas de la imagen en el flujo de bits comprimido.
- Poder acelerar las diferentes partes del algoritmo en hardware que se adecúe a las operaciones realizadas en cada una.

En la Figura 5.1 podemos ver el esquema general del algoritmo desarrollado, que se ha llamado "Jypec" (Java Hyperspectral Compressor). Consta de varias fases, que transforman los datos originales en crudo en un flujo de bits comprimidos:

- Se aplica una reducción dimensional en la dirección espectral, reduciendo el número de bandas necesarias para guardar la imagen. En este paso se elimina información redundante o repetitiva, perdiéndose información poco relevante de cara a las características más generales de la imagen.

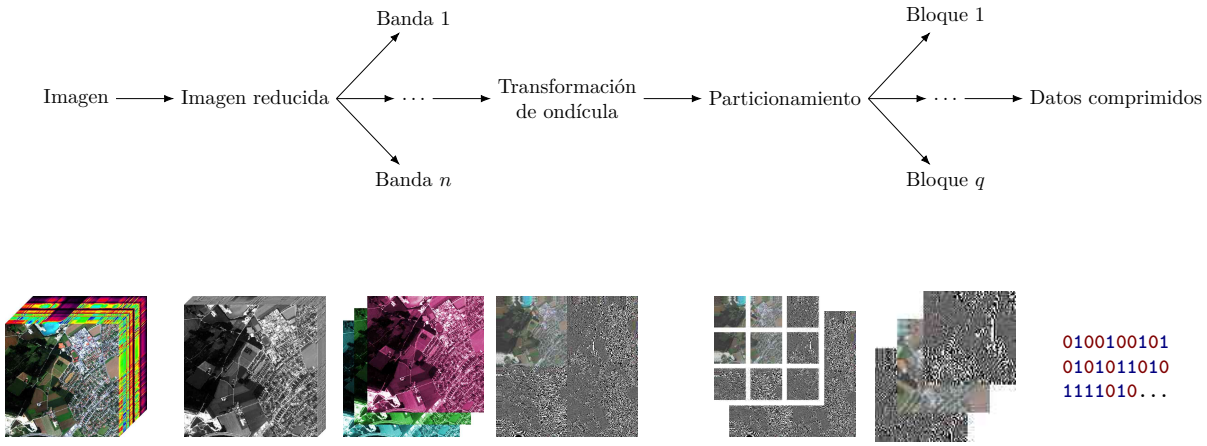


Figura 5.1: Esquema general del funcionamiento del algoritmo. Poco a poco se fragmenta la imagen original en pedazos hasta que pueden ser comprimidos.

- Las bandas se tratan como elementos independientes, y a cada una se le aplica, opcionalmente, una transformación de ondícula para concentrar la información restante, pudiendo comprimir más agresivamente después estas bandas transformadas.

- A continuación se divide cada banda en bloques, etiquetados según el sector de la ondícula al que pertenecieran (si la hubo), lo cual ayuda aún más en la compresión.

- Los bloques son codificados independientemente, y se concatenan uno tras otro en un orden predefinido.

5.1. Consideraciones a la hora de implementar

- El objetivo desarrollando este algoritmo es el de conseguir un sistema de compresión de imágenes hiperespectrales, con pérdida, capaz de mantener una buena calidad aun cuando se realizan compresiones agresivas.

- No se pretende generar un flujo que pueda ser incrementalmente decodificado como en JPEG2000 con el principio EBCOT[1] [55]. La idea es ajustar unos parámetros de compresión, y obtener un archivo del tamaño lo más reducido posible, eliminando todo tipo de marcadores.

- No se utilizará el sistema de reducción dimensional proporcionado por el estándar JPEG2000. Se quieren probar métodos de reducción dimensional que no encajan con los tres patrones predefinidos por JPEG2000. Por tanto se creará un formato propio para mantener una justa comparativa.

- Se utilizará la transformación de ondícula con pérdida proporcionada por JPEG2000.

- Se utilizará el codificador aritmético binario definido en el JPEG2000, sin cambios en los modelos predictivos.

[1]EBCOT, o codificador embebido de bloques con truncado óptimo, busca generar un flujo de datos comprimido donde la decodificación de cualquier prefijo ofrezca una aproximación progresivamente más refinada de los datos originales

5.2. Lectura de los datos

El primer paso que debemos llevar a cabo es leer los ficheros de entrada para cargar nuestra imagen en memoria. La lectura de la imagen se realiza completa antes de pasar a comprimir.

Existen dos partes diferenciadas en la lectura: La lectura de la cabecera de la imagen (metadatos) y la lectura de los datos en sí. Siempre se leen antes los metadatos, pues nos dan información acerca del formato y ordenación de los datos, y sería imposible leer estos últimos sin dicha información.

```
ENVI
description = {
    Test image hdr
}
samples = 65
bands = 6
lines = 40
byte order = 0
data type = 12
interleave = BIL
wavelength = {
    365.9298,
    375.5940,
    385.2343,
    395.1235,
    405.0023,
    414.9885
}
```

Figura 5.2: Ejemplo de archivo simple de cabecera ENVI.

5.2.1. Lectura de los metadatos

El formato de cabecera de las imágenes hiperespectrales utilizadas es el *ENVI header* (`.hdr`). Un archivo `.hdr` es un conjunto de parejas **<clave>** = **<valor>**, separadas por saltos de línea. En ocasiones el valor ocupa varias líneas, en cuyo caso es encerrado en llaves {**<valor>**}. El archivo comienza siempre con la cadena ascii de 5 bytes "`ENVI\n`", que identifica el tipo de archivo. A partir de ese punto comienza la colección de parejas clave-valor. Podemos ver un ejemplo en la Figura 5.2.

Existen varias decenas de claves [56], algunas con formatos específicos como fechas, información sobre el sistema de coordenadas utilizado, posición sobre el globo terráqueo de la imagen, etc. Para la lectura de los datos sólo nos hacen falta unos cuantos metadatos concretos, por lo que necesitaremos saber tan solo el formato específico de unos pocos valores, pudiendo ignorar los demás:

- **bands**: Número entero positivo que indica el número de bandas en la imagen.
- **lines**: Número entero positivo que indica el número de líneas en la imagen. (La altura de la misma).
- **samples**: Número entero positivo que indica el número de muestras por línea en la imagen. (La anchura de la imagen).
- **byte order**: Un `0` nos indica un formato *little endian*, donde los bytes de cada muestra se ordenan de menos a más significativos. Un `1` indica formato *big endian*.
- **data type**: Tipo enumerado que indica el formato de los datos: Números enteros o en coma flotante, cantidad de bits utilizados para representarlos, o si se utiliza o no signo.
- **interleave**: Enumerado que indica el orden de las muestras de la imagen. Existen Intercalado secuencial por banda (BSQ), Intercalado por píxel y banda (BIP) y Intercalado por línea y banda (BIL).
- **header offset**: Los datos y metadados pueden estar en el mismo archivo o en diferentes. En el primer caso, el valor de este campo nos indica el número de bytes a saltar desde el principio del fichero para llegar a la zona de datos (que siempre viene después de los metadatos).

A la hora de interpretar el fichero para leer los parámetros, hacemos uso de expresiones regulares para separar por completo todos los pares clave-valor. Las expresiones regulares nos

permiten separar los patrones de **<clave>** = **<valor>** detectando cada elemento de manera independiente.

5.2.1.1. Metadatos en el archivo comprimido

Estas parejas están en texto plano, tanto las claves como los valores. Para leer nos es cómodo, ya que dada una clave sabemos fácilmente cual es el tipo de los datos y podemos parsearlos sin problema. Sin embargo es muy poco eficiente a nivel de memoria. Guardar una decena de bytes para una clave, cuando solo existen unas cincuenta diferentes es un desperdicio, pudiendo utilizar un único byte para identificarlas.

Por tanto, para la escritura de los datos en el flujo comprimido, las claves se traducen a un tipo enumerado que se guarda en un único byte.

Los valores también son procesados y guardados en el formato más óptimo para su tipo:

- **Valores discretos**: En ocasiones un valor solo tiene un número finito de opciones. En ese caso se calcula el número mínimo de bytes para representar todas las posibilidades, y se utiliza ese tamaño para el campo.
- **Cadenas de texto**: No queda más remedio que guardarlas en crudo, pues suelen ser descripciones de la imagen que no pueden inferirse ni saberse a priori.
- **Números**: Los números se parsean y guardan en crudo. Un entero ocupará cuatro bytes, una flotante de precisión doble ocho, etc.
- **Números pequeños**: En caso de enteros que sepamos que suelen ser de pequeño tamaño, se utilizará un formato de longitud variable. En este caso cada byte tiene únicamente siete bits útiles, utilizando el último para indicar si el siguiente byte también forma parte del número en cuestión.
- **Arrays**: Los arrays de un tipo concreto de datos, en formato texto, tienen numerosas comas y espacios que no aportan información. La forma de comprimirlos es guardar un entero que nos diga el número de elementos a esperar, seguido de los datos en crudo.
- **Tipos nativos**: Otros tipos nativos, como los caracteres de texto se guardarán en crudo con el número de bytes que les corresponda.

Como ejemplo, la cabecera mostrada al principio de la Sección 5.2.1 ocupa 223 bytes. Comprimida con el formato comentado pasa a ocupar tan solo 69:

```
FF 01 54 65 73 74 20 69 6D 61 67 65 20 68 65 61     ÿ.Test image hea
64 65 72 02 00 00 00 41 03 00 00 00 06 04 00 00     der....A........
00 28 05 00 06 0C 07 01 08 00 00 00 06 43 B6 F7     .(.............C¶÷
04 43 BB CC 08 43 C0 9D FE 43 C5 8F CF 43 CA 80     .C»Ì.CÀ.-CÅ.ÏCÊ€
4B 43 CF 7E 87                                      KCÏ ‡
```

Observamos las secciones de la cabecera empaquetadas una tras otra. Los bytes azules indican la clave, y el valor (cuyo tamaño se conoce pues es fijo) viene inmediatamente después. En caso de valores con tamaño variable, como arrays, se añade la longitud inmediatamente después de la clave (azul verdoso).

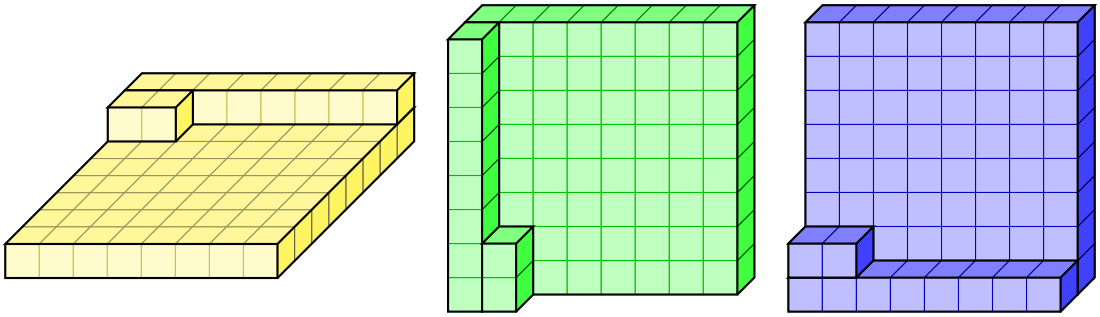


Figura 5.3: Las tres formas principales de recorrer una imagen: BSQ, BIP y BIL respectivamente.

5.2.2. Lectura de los datos - EJML

Tras leer los metadatos, tenemos ya la información suficiente para leer los datos. Estos podrán estar a continuación en el mismo archivo, o en otro archivo diferente, que en este caso tiene el mismo nombre que el archivo de metadatos, pero sin la cabecera `.hdr`.

Para leer los datos, nos fijamos en las dimensiones de la imagen, la ordenación de las muestras en el fichero (BSQ, BIL, BIP (ver Figura 5.3)) y el formato de las muestras (Número de bytes, tipo de datos y orden big o little endian).

La imagen completa se carga en una estructura de matriz bidimensional. Cada columna es un pixel hiperspectral (con tantas muestras como bandas). El píxel de la esquina superior izquierda de la imagen es el primero, y a partir de ahí se van recorriendo por líneas hasta llegar al último, que es el de la esquina inferior derecha.

La estructura de datos que guarda las imágenes leídas es `FMatrixRMaj`, una matriz con ordenación *row-major* de la biblioteca de java EJML [57] (Efficient Java Matrix Library) orientada al alto rendimiento en procesado de matrices.

Además de utilizar las estructuras de datos de esta biblioteca, se utilizarán algunos algoritmos de extracción de autovalores, autovectores y valores singulares.

Junto al tipo de datos `FMatrixRMaj`, de precisión flotante simple, se ofrece `DMatrixRMaj`, análogo para precisión flotante doble. El uso de este segundo se ha descartado por dos motivos: consumir el doble de memoria y además no ser necesario, debido a que trataremos habitualmente con números de 16 bits, que tienen representación exacta en coma flotante simple.

5.3. Reducción de dimensionalidad

Se han implementado como algoritmos de reducción dimensional **ICA**, **MNF**, **PCA**, **SVD**, **VCA** y **VQPCA** (ver sección 3.6). Pese a las diferencias entre ellos, comparten un flujo común entre todos:

- `preprocess`: Preprocesa la matriz, por ejemplo para extraer un subconjunto de muestras con las que entrenar, para simplificar el proceso de entrenamiento. La matriz devuelta tiene el mismo número de columnas (bandas) pero menor número de filas (muestras).

- `train`: Partiendo de la imagen original, entrena el modelo de reducción dimensional con la información de todos los píxeles dados. En este paso se generan las matrices de proyección, vectores de media para el centrado, y otras estructuras que pueda necesitar el método de reducción en cuestión.

- `reduce`: Reduce la dimensión de los datos de entrada. La nueva matriz tiene el mismo número de filas que la original, pero éstas son de menor longitud, y representan cada una a la correspondiente fila de la original en el espacio reducido.

- `boost`: Realiza el proceso inverso a `reduce`, tomando las filas en el espacio reducido, y devolviéndolas mediante aproximación al espacio original.

Centrándonos en la compresión, las tres primeras funciones se utilizan al comprimir, y la cuarta al descomprimir. Es claro que si volcamos los archivos a disco, al descomprimir no vamos a tener en memoria las matrices de proyección y demás elementos necesarios para volver al espacio original.

Por ello todos los métodos de reducción cuentan con funciones de guardado y recuperación en flujos de bits, para poder empotrar en el archivo comprimido los datos necesarios que permitan la descompresión. Nótese que nunca se guarda información necesaria exclusivamente para comprimir, pues hacerlo sería un desperdicio de espacio.

Así, salvo excepciones comentadas posteriormente en sus secciones, el procesado completo genera los siguientes elementos:

- Una matriz W, que transforma los datos de entrada X en los transformados $T = XW$.

- Una segunda matriz $\bar{W}$, que hace la transformación inversa $X = T\bar{W}$.

- Opcionalmente, un vector $\bar{\boldsymbol{x}}$ que centra los datos de entrada antes de aplicar W. Se utiliza pues las transformaciones W y $\bar{W}$ suelen funcionar mejor en datos centrados en cero, así que sustraer la media como preprocesamiento es una buena idea.

De los cuales necesitaremos guardar $\bar{W}$ y $\bar{\boldsymbol{x}}$ para hacer la descompresión, mientras que W podrá descartarse tras comprimir.

$\bar{W}$ es una matriz de tamaño $m \times n$, y $\bar{\boldsymbol{x}}$ es un vector de longitud n. Esto supone el sobrecoste de guardar, además de los datos comprimidos, un total de $(m+1)n$ números.

En este caso todas las operaciones se realizan en coma flotante. La razón es que los números con los que se trata habitualmente suelen ser enteros de 16 bits reconvertidos a coma flotante, y utilizar aritmética de precisión doble en la conversión resulta en un sobrecoste muy grande respecto a la precisión simple, sin existir diferencia apreciable en la calidad del resultado.

Por tanto, el total de los metadatos en cuanto a reducción dimensional ascenderá a $4(m+1)n$ bytes, insignificante frente al tamaño de la matriz T de datos reducidos, que inevitablemente habrá que guardar.

5.3.1. Implementaciones concretas

5.3.1.1. Proyecciones lineales

Los métodos **ICA**, **MNF PCA**, **SVD** y **VCA** utilizan todos proyecciones lineales para su ejecución, y por tanto comparten las funciones `reduce` y `boost` (Algoritmos 1 y 2):

El Algoritmo 3 se omite en **VCA**, que solo hace la proyección. El resto sí hacen el proceso completo. En cuanto a la función de entrenamiento, ahí sí difieren todos estos algoritmos, y es lo que se explica en los siguientes apartados.

Algoritmo 1: DimReduction.reduce

input : Una matriz $X \in \mathcal{M}_{n \times p}$ con p datos para reducir de dimensión n
input : Un ajuste $\bar{\boldsymbol{x}}$ para centrar X
input : Una matriz de proyección $W \in \mathcal{M}n \times m$
output: Una matriz $T \in \mathcal{M}_{m \times p}$ de datos reducidos

1 $X \leftarrow X - \bar{\boldsymbol{x}}$;
2 $T \leftarrow (X^T * W)^T$;

Algoritmo 2: DimReduction.boost

input : Un ajuste $\bar{\boldsymbol{x}}$ para centrar X
input : Una matriz de proyección inversa $\bar{W} \in \mathcal{M}m \times n$
input : Una matriz $T \in \mathcal{M}_{m \times p}$ de datos reducidos
output: Una matriz $X \in \mathcal{M}_{n \times p}$ con los datos originales

1 $X \leftarrow (T^T * \bar{W})^T$;
2 $X \leftarrow X + \bar{\boldsymbol{x}}$;

5.3.1.2. ICA

ICA trabaja sobre datos centrados en cero, así que el primer paso pasa por extraer la media y centrar en cero. Acto seguido los datos se empaquetan y envían a la API de JSAT [58] que se encarga de aplicar el algoritmo FAST ICA [59, 60]. El algoritmo FAST ICA está basado en funciones que permiten tanto la estimación de la descomposición global en componentes independientes, como la estimación de componentes individuales. Todas las estimaciones se basan en un modelo lineal para la mezcla de las señales, consiguiendo gran rapidez en los cálculos.

Tras la llamada a la API, los datos devueltos (matriz de proyección y recuperación) se traducen al formato entendible por nuestra aplicación Sección 5.2.2.

5.3.1.3. MNF

La fracción mínima de ruido consta de dos etapas diferenciadas: La estimación del ruido, y el cálculo de la matriz de proyección.

La parte más subjetiva es la estimación del ruido. Lo ideal sería partir de una medida real del ruido, pero lo más parecido que podemos conseguir es una estimación, asumiendo una cierta correlación espacial entre valores, siguiendo la Ecuación (3.19).

Una vez obtenido el ruido, se calculan las matrices de covarianza tanto de la señal como del ruido. Si bien la definición requiere la resolución de un problema de descomposición en valores singulares generalizado según la Ecuación (3.21), la implementación subyacente resuelve dos SVD, siguiendo el método propuesto por [61].

Algoritmo 3: DimReduction.center

input : Una matriz $X \in \mathcal{M}_{n \times p}$ con p datos para reducir de dimensión n
output: La media $\bar{\boldsymbol{x}}$ de X
output: X centrado sobre la media

1 $\bar{\boldsymbol{x}} \leftarrow$ la media de X;
2 $X \leftarrow X - \bar{\boldsymbol{x}}$;

Algoritmo 4: MNF.train

input : Una matriz $X \in \mathcal{M}_{n \times p}$ con p datos de entrenamiento de dimensión n
input : La dimensión m objetivo
output: Las matrices W y $\bar{W}$ de reducción y aumento de dimensionalidad

1 $\mathcal{N} \leftarrow$ estimación de ruido obtenida de X;
2 $S_{\mathcal{N}} \leftarrow$ matriz de covarianza del ruido $\mathcal{N}$;
3 $S \leftarrow$ matriz de covarianza de los datos X;
4 $W \leftarrow$ es la colección de autovectores resultantes de resolver el problema SVD generalizado $\boldsymbol{w} S_{\mathcal{N}} = \lambda \boldsymbol{w} S$, quedándonos con los m de mayor autovalor asociado;
5 $\bar{W}$ se calcula como la inversa de W;

5.3.1.4. PCA

PCA comienza con un centrado de los datos en cero, para posteriormente generar su matriz de covarianza. Extraemos los autovectores de la matriz de covarianza y los colocamos en forma de matriz, sirviéndonos ésta para cambiar a la base de los autovectores. Al ordenarlos de mayor a menor autovalor, conseguimos que las primeras direcciones de la nueva base (direcciones principales) sean aquellas de mayor varianza en los datos originales. Basta por tanto quedarse con las m primeras direcciones principales para obtener una buena aproximación de los datos originales al deshacer la transformación, como se ve en el Algoritmo 5.

Algoritmo 5: PCA.train

input : Una matriz $X \in \mathcal{M}_{n \times p}$ con p datos de entrenamiento de dimensión n
input : La dimensión m objetivo
output: Las matrices W y $\bar{W}$ de reducción y aumento de dimensionalidad

1 `center(`X`);`
2 $S \leftarrow$ matriz de covarianza de los datos X;
3 $W \leftarrow$ autovectores de S;
4 $\Lambda \leftarrow$ autovalores asociados a W de S;
5 $W \leftarrow$ es el resultado de ordenar W según Λ de mayor a menor, quedándonos con los m primeros autovectores;
6 $\bar{W} \leftarrow$ es igual a la transpuesta W^T;

5.3.1.5. SVD

Tras opcionalmente centrar los datos, se procede a llamar a la API de EJML [57] para obtener la descomposición SVD. Los vectores devueltos por EJML no están ordenados, así que hacemos lo mismo que en PCA y los reordenamos para quedarnos con los que más información contienen sobre los datos originales (Algoritmo 6).

5.3.1.6. VCA

VCA sigue la implementación definida en [41], pero utilizando únicamente SVD para el preprocesado, ya que con el algoritmo propuesto para PCA no se conseguían buenos resultados. Tras aplicar SVD a los datos de entrada, se van generando iterativamente los miembros extremos del conjunto, en función de los cuales se representarán los demás en la reducción (Algoritmo 7).

Algoritmo 6: SVD.train

input : Una matriz $X \in \mathcal{M}_{n \times p}$ con p datos de entrenamiento de dimensión n
input : La dimensión m objetivo, y una opción `center` que indica si queremos centrar W
output: Las matrices W y $\bar{W}$ de reducción y aumento de dimensionalidad

if `center` **then**
    `center(`X`);`
end
Realizar la descomposición en valores singulares $X = U\Lambda V^T$;
W es el resultado de ordenar los autovectores de U descendentemente según los valores de la diagonal de Λ;
$\bar{W} \leftarrow$ es igual a la transpuesta W^T;

Algoritmo 7: VCA.train

input : Una matriz $X \in \mathcal{M}_{n \times p}$ con p datos de entrenamiento de dimensión n
input : La dimensión m objetivo
output: Las matrices W y $\bar{W}$ de reducción y aumento de dimensionalidad

`svd` $\leftarrow$ new `SingularValueDecomposition(`m`);`
$W_{svd} \leftarrow$ `svd.train(`X`);`
$X_{svd} \leftarrow$ proyección de X sobre W_{svd};
$Y \leftarrow$ obtenido de dividir X_{svd} entre su proyección sobre la media de W_{svd};
Inicializar el conjunto A al primer vector de la base canónica;
Inicializar X_s al conjunto vacío;
for $i \leftarrow 0$ **to** $m - 1$ **do**
    $\boldsymbol{f} \leftarrow$ vector aleatorio ortogonal al espacio abarcado por A;
    $V \leftarrow$ producto escalar de los elementos de Y contra $\boldsymbol{f}$;
    Añadir al espacio vectorial A el elemento de Y con mayor V asociado;
    Añadir a los extremos X_s el elemento de X con mayor V asociado;
end
W es el resultado de proyectar X_s sobre W_{svd};
$\bar{W}$ es la pseudoinversa de W;

5.3.1.7. Proyección no lineal - VQPCA

Por último, tras todos los métodos lineales se encuentra VQPCA, que se categoriza dentro de los no lineales.

En resumen, se realiza un clustering de los datos de entrada, y cada cluster se procesa como un PCA independiente (ver Algoritmos 8 a 10).

El clustering se hace mediante dos librerías, a elegir: SMILE [62] o JSAT [58] que utilizan ambas el método K-Means [63-65].

- **JSAT** realiza un clustering inicial sobre el total de los datos de entrenamiento, y no permite averiguar posteriormente el cluster de una muestra. Así, la matriz T^C del Algoritmo 9 tiene que generarse en el Algoritmo 8, y por tanto los datos de entrenamiento deben ser iguales a los que serán reducidos posteriormente. Esto es un problema menor en cuanto a precisión, pero sí supone un problema grande en cuanto a memoria, al no poder reducirse el conjunto de entrenamiento.
- **SMILE** por contra sí permite el entrenamiento con un conjunto reducido, haciendo una aproximación de los clusters menos exacta, pero mejorando notablemente en velocidad.

Algoritmo 8: VQPCA.train

input : Una matriz $X \in \mathcal{M}_{n\times p}$ con p datos de entrenamiento de dimensión n
input : La dimensión m objetivo
input : El número c de clusters deseados
output: Un clasificador C para asignar cada muestra a uno de los c clusters
output: m algoritmos `PrincipalComponentAnalysis` para reducir independientemente cada cluster

```
1 C ← clasificador K-Means entrenado con X;
2 Inicializar pcas al conjunto vacío;
3 for i ← 1 to c do
4     X_i ← cluster iésimo resultante de aplicar C a X;
5     pca ← new PrincipalComponentAnalysis(m);
6     pca.train(X_i);
7     Añadir pca a pcas;
8 end
```

Algoritmo 9: VQPCA.reduce

input : Una matriz $X \in \mathcal{M}_{n\times p}$ con p datos para reducir de dimensión n
input : Un clasificador C con c clusters, y c algoritmos `PrincipalComponentAnalysis`
output: Una matriz $T \in \mathcal{M}_{m\times p}$ de datos reducidos
output: Una matriz T^C indicando el cluster de cada $t \in T$

```
1 T ← ∅;
2 T^C ← ∅;
3 for cada muestra x en X do
4     Encontrar el cluster i al que pertenece x utilizando C;
5     t ← es el resultado de aplicar el pca iésimo a x;
6     Añadir t a T;
7     Añadir i a T^C;
8 end
```

Algoritmo 10: VQPCA.boost

input : Una matriz $T \in \mathcal{M}_{m\times p}$ con p datos reducidos de dimensión m
input : pcas: c algoritmos `PrincipalComponentAnalysis`, y una matriz T^C indicando el pca aplicado a cada $t \in T$
output: Una matriz $X \in \mathcal{M}_{n\times p}$ de datos reconstruidos

```
1 X ← ∅;
2 for cada muestra t e índice t^i en (T, T^C) do
3     x ← es el resultado de invertir pcas [t^i] sobre t;
4     Añadir x a X;
5 end
```

5.4. Detección de puntos aislados

Tras la reducción dimensional, puede que algunas muestras sean muy extremas a nivel de banda. Para la transformación de ondícula influye la amplitud del intervalo entre el mínimo y máximo de las muestras, y lo ideal es que sea lo más pequeño posible. Cuanto más grande sea, menor separación habrá posteriormente en las muestras transformadas, y por tanto más fácil será que los redondeos hagan iguales dos muestras que en origen no lo eran.

Para mitigar estos problemas, se puede establecer un porcentaje de las muestras extremas que se guardará en crudo en el archivo comprimido, con el formato (valor, coordenada x, coordenada y). Los elementos aislados se truncarán al máximo (o mínimo) no aislado, y se continuará con el siguiente paso como si hubieran tenido ese valor desde el origen (ver Figura 5.4).

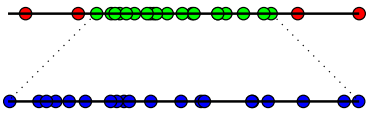


Figura 5.4: Más resolución para los puntos no aislados.

5.5. Transformación de ondícula

Como se veía en la sección 4.2.2, la principal función de las ondículas es transformar un conjunto de datos para concentrar la información de los mismos, y poder aplicar una compresión más agresiva en las partes que menos información contienen.

Recordemos que los datos resultantes de la reducción están guardados en la matriz T:

$$T = [\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p], \quad \boldsymbol{t}_i \in \mathbb{R}^m$$

La ondícula la aplicamos a nivel de banda, por tanto vamos a dividir T en las bandas $B^k, k \in \{1, \ldots, m\}$:

$$B^k = \left[\boldsymbol{t}_1^k, \ldots, \boldsymbol{t}_p^k\right] = [b_1, \ldots, b_p]$$

Aunque hasta ahora hemos estado hablando de los datos (X, T, B) como listas de muestras, es también útil interpretarlos como matrices, que es la representación natural de las imágenes. Así, el tamaño de la imagen es de $w \times h \times b$ (anchura, altura y bandas), donde $w \times h = p$, y $b = n$ para X, $b = m$ para T. Por tanto podemos definir:

$$B^k = \left[\boldsymbol{t}_{i,j}^k\right] = [b_{i,j}], \quad i \in \{1, \ldots, h\}, j \in \{1, \ldots, w\} \quad b_{i,j} = b_{i*w+j}$$

A las filas y columnas de B^k se les aplican (ver Figura 5.5) los dos filtros de una ondícula (paso alto y bajo) en forma de kernels (Definición 3). En este caso los kernels aplicados son:

$$K_h = \begin{bmatrix}0{,}026 & -0{,}016 & -0{,}078 & 0{,}266 & 0{,}602 & 0{,}266 & -0{,}078 & -0{,}016 & 0{,}026\end{bmatrix} \tag{5.1}$$

$$K_l = \begin{bmatrix}0{,}091 & -0{,}057 & -0{,}591 & 1{,}115 & -0{,}591 & -0{,}057 & 0{,}091\end{bmatrix} \tag{5.2}$$

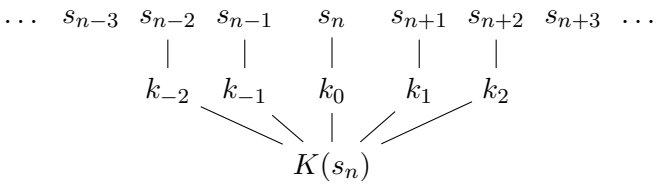


Figura 5.5: Aplicación de un kernel sobre una secuencia. Se mantiene una ventana deslizante sobre la que se opera.

La transformación de ondícula representada por estos kernels se conoce como CDF 9/7 [66].

Donde los vectores $\boldsymbol{v} = [v_1, \ldots, v_n] \in \mathbb{R}^n$ de entrada se extienden en los bordes de la siguiente forma:

$$v_i = \begin{cases} v_i & 1 \leq i \leq n \\ v_{2-i} & i < 1 \\ v_{n-(i-n)} & i > n \end{cases} \tag{5.3}$$

De la aplicación de los dos kernels obtenemos:

$$K_h(\boldsymbol{v}) = \boldsymbol{v}^h \quad K_l(\boldsymbol{v}) = \boldsymbol{v}^l$$

A partir de estos resultados montamos el vector final resultante de aplicar la transformación, que queda como:

$$\boldsymbol{v}' = \left\{ \left\{ \boldsymbol{v}_i^l \right\}_{i \in 2\mathbb{N}}, \left\{ \boldsymbol{v}_i^h \right\}_{i \in 2\mathbb{N}} \right\}$$

El mismo proceso se puede aplicar de nuevo sobre la primera mitad de v' recursivamente, consiguiendo concentrar cada vez más la información en los primeros valores del vector. Habitualmente se aplica entre una y cuatro veces.

Tras pasar todas las filas de B^k por este proceso, se vuelve a aplicar análogamente por columnas, consiguiendo finalmente el resultado que veíamos en la Figura 4.8.

La forma evidente de aplicar los kernels mostrada en la Figura 5.5 tiene un coste de $n \cdot k$, donde n es la longitud de la secuencia y k la del kernel. Dado que los coeficientes son simétricos ($k_{-n} = k_n$), se ha ideado un método de cálculo que reduce el número de operaciones, conocido como *lifting* [67]. Es el que se utilizará aquí, por ofrecer una mejora en tiempo de aproximadamente el 50 % para el filtro CDF97. En resumen, en lugar de calcular elemento a elemento el resultado, hace una serie de operaciones a nivel de vector, del orden de la mitad de la longitud del filtro, como se observa en la Figura 5.6.

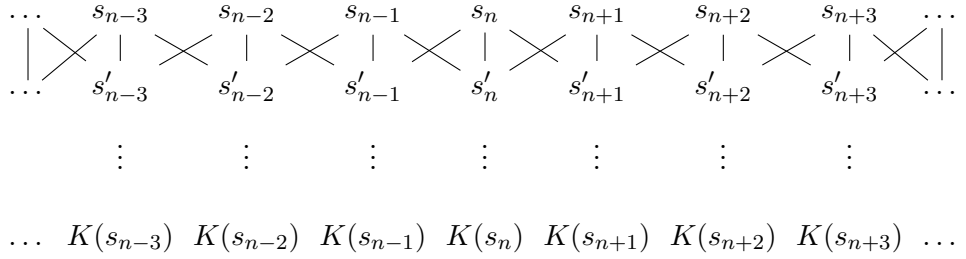


Figura 5.6: Aplicación de un kernel sobre una secuencia utilizando la técnica de lifting. Se mantiene una ventana deslizante sobre la que se opera.

5.6. Cuantización

Hasta ahora, todas las etapas se han realizado en números de coma flotante de 32 bits. La razón es que los datos originales suelen ser muestras enteras de 16 bits sin signo, y trabajar directamente con ellos sería infactible para tareas como la reducción dimensional, por lo que se transforman a números decimales.

Para etapas posteriores vuelve a ser necesaria la representación en números enteros, por lo que hay que cuantizar los decimales para transformarlos al formato deseado.

La cuantización traslada números de un intervalo real a otro mediante interpolación lineal, y redondea al entero más cercano. En nuestro caso el intervalo original está acotado matemáticamente al $(-1, 1)$. El intervalo de salida dependerá de la calidad que queramos en la imagen comprimida, pero será de la forma $[-2^n + 1, 2^n - 1]$. Basta tomar la muestra original, multiplicar por 2^n y redondear hacia el cero para realizar la cuantización.

5.6.1. Funciones de precuantización

Si sabemos que nuestros datos tienen una distribución sesgada dentro del intervalo (-1, 1), podemos aplicarles una función biyectiva en el intervalo que los separe (o junte) para que los redondeos afecten menos a nuestras muestras (por ejemplo si todos están cerca del cero, separarlos hacia los extremos).

Las funciones de precuantización centran los datos en cero utilizando la media para posteriormente estirar o encoger el intervalo que ocupan. Si denotamos a la media $\boldsymbol{x}$, algunas funciones de precuantización son:

$$f_{log}(x) = \begin{cases} \log(x - \boldsymbol{x} + 1) & x > \boldsymbol{x} \\ \log(\boldsymbol{x} - x + 1) & x < \boldsymbol{x} \\ 0 & x = 0 \end{cases} \qquad f_{sqrt}(x) = \begin{cases} \sqrt{x - \boldsymbol{x}} & x > \boldsymbol{x} \\ \sqrt{\boldsymbol{x} - x} & x < \boldsymbol{x} \\ 0 & x = 0 \end{cases} \qquad f^r_{lin}(x) = \frac{x}{r}$$

5.7. EBCoding

Tras cuantizar, pasamos a dividir la imagen en bloques de 64×64 (o menos si no se puede dividir exactamente), que se codifican independientemente realizando varias pasadas por sus diferentes planos de bits, siguiendo la sección 4.2.4. En cada una de sus pasadas (significancia, refinamiento y limpieza) se hace un recorrido en zig-zag por columnas, abarcando cuatro filas cada vez.

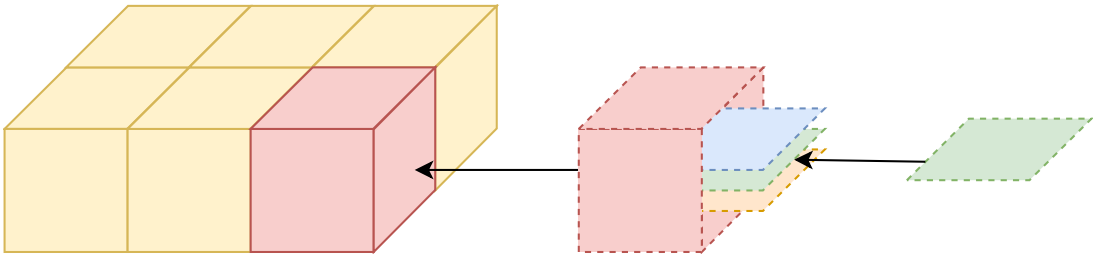


Figura 5.7: La imagen reducida y cuantificada (izda) se divide en bloques. Cada bloque no duplica los datos, sino que referencia los originales. De la misma manera, cada plano de bits referencia al bloque. Las referencias se deshacen internamente al recibir la petición de un dato.

Para ahorrar en memoria, los bloques y los planos guardan referencias a los datos originales ya transformados y cuantizados (Figura 5.7). Los datos sobre la significancia de cada muestra en el bloque, y el indicador de haber sido refinada se guardan aparte, en el propio codificador, para evitar introducir información innecesaria en los bloques y planos, que sólo representan datos.

Además de los datos en crudo, necesitamos el contexto (Definición 7) de cada bit. Para su extracción tenemos dos opciones. Una es mirar el estado de significancia y realizar operaciones de conteo (que tendremos que hacer para cada muestra en todas las pasadas por los planos). La otra es precalcular el conteo y así mirar de forma más directa el contexto que recuperamos. (Ver Figura 5.8).

El contexto tiene una peculiaridad, y es que no siempre se calcula de la misma manera. Dependiendo de la pasada en la que estemos, y de la subbanda a la que pertenezca el bloque, se llamará a una u otra función de generación de contextos [29] para cada bit. No obstante siempre dependerá del estado de significancia de un vecindario de 3×3 alrededor del bit que está siendo codificado.

El Algoritmo 11 muestra las tres pasadas dadas en el bloque para codificarlas, excepto en el

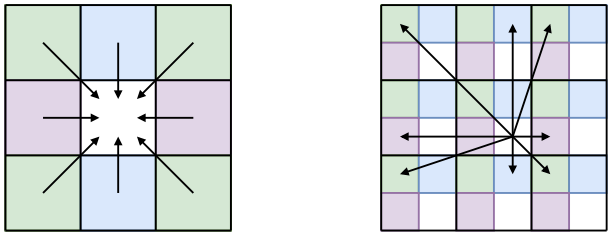


Figura 5.8: El contexto depende de los vecinos diagonales (verde), horizontales (morado) y verticales (azul). A la izquierda, se calcula mirando directamente dichos vecinos. A la derecha, cada vez que se actualiza la significancia de una muestra se actualizan los conteos de los vecinos, con lo que se aceleran los cálculos posteriores.

Algoritmo 11: EBCoder.code

input : Un bloque de la imagen con n_p planos de bits
input : Un bitStream donde verter los datos codificados
input : Un codificador aritmético binario mqCoder

```
1  for i ← 0 to n_p − 1 do
2      plano ← bloque.getPlane(i);
3      if i ≠ 0 then
4          significancePass(plano, bitStream);          /* Algoritmo 12 */
5          refinementPass(plano, bitStream);            /* Algoritmo 13 */
6      end
7      cleanupPass(plano, bitStream);                   /* Algoritmo 14 */
8  end
9  Completar bitStream a un número de bits múltiplo de 8;
10 Añadir el código de terminación de bloque a bitStream;
```

primer plano donde solo se hace la limpieza. Al final se completa a un número entero de bytes, ya que el codificador aritmético, llamado internamente, genera secuencias de bits de longitud arbitraria. El código de terminación se añade como marcador para el decodificador.

En el Algoritmo 12 observamos cómo las muestras solo se codifican en la pasada de significancia una vez (pues una vez significantes no lo dejarán de ser). En el caso especial de codificar un `1`, se manipula y codifica el signo utilizando un bit de XOR asociado al contexto, a fin de mejorar la compresión.

La simplicidad de la pasada de refinamiento se ve en el Algoritmo 13, donde todos los bits significantes que no hayan sido aún codificados se mandan junto a su contexto al codificador MQ. No nos preocupamos del signo pues ya habrá sido codificado al hacer la muestra significante.

Por último, en el Algoritmo 14 se hace la etapa más complicada: limpieza. Aquí se intenta hacer una codificación de carrera (Sección 3.5) cuando se encuentran ceros seguidos en una columna de 4 muestras. En caso de fallar esta codificación de carrera, se entra en un modo de "emergencia" que indica la posición del fallo y a continuación codifica siguiendo el Algoritmo 12.

5.8. MQCoding

A lo largo de los Algoritmos 12 a 14 hemos visto diferentes llamadas al codificador MQ. La responsabilidad del EBCoder era la de generar parejas (bit, contexto), y es ahora el codificador aritmético el encargado de generar los bits comprimidos a partir de la secuencia de parejas generada.

Algoritmo 12: EBCoder.codeSignificance

```
1  for cada bit b en el recorrido en zig-zag de plano do
2      context ← contexto de significancia de b;
3      if b no es significante, y context ≠ ContextZERO then
4          mqCoder.code(b, context, bitStream);
5          if b es el bit 1 then
6              b_s ← bit de signo asociado a b;
7              context ← contexto de signo de b_s;
8              x_s ← bit de XOR asociado a context;
9              mqCoder.code(b_s ⊕ x_s, context, bitStream);
10         end
11         Marcar b como ya codificado en plano;
12     end
13 end
```

Algoritmo 13: EBCoder.codeRefinement

```
1  for cada bit b en el recorrido en zig-zag de plano do
2      if b no está codificado, y es significante then
3          context ← contexto de refinamiento de b;
4          mqCoder.code(b, context, bitStream);
5          Marcar b como ya codificado en plano;
6      end
7  end
```

La implementación del codificador aritmético es muy ligera (Algoritmo 15), pues la parte costosa de la codificación en cuanto a espacio (estado de significancia y demás) ya se la queda el EBCoder.

El codificador aritmético guarda una serie de variables: Los registros A y C (Sección 4.2.5) que indican respectivamente la longitud y límite inferior del intervalo que se está subdividiendo, así como una variable $\bar{t}$ con la cantidad de bits consolidados en C, y un buffer temporal T para evitar generar secuencias de bytes reservadas.

Mantiene además una tabla de entradas (contexto, (bit, estado)) donde se guarda, asociado a cada contexto:

- Un bit de predicción, que indica el bit que el codificador MQ espera que venga asociado al contexto asociado. Cuando esta predicción es certera, la eficiencia de codificación aumenta, y se gastan menos bits.
- Un estado que, mediante tablas estáticas, indica la probabilidad de que se dé la predicción, así como transiciones a otros estados según vayan llegando nuevos símbolos.

Las tablas estáticas necesarias son las siguientes:

- Tabla de transición para el símbolo más probable. Indica el estado al que cambiar si la predicción es certera.
- Tabla de transición para el símbolo menos probable. Análoga a la anterior pero indicando a qué estado cambiar cuando falla la predicción.

Algoritmo 14: EBCoder.codeCleanup

```
1  for cada bit b en el recorrido en zig-zag de plano do
2      if b es el primero de su columna de 4, y toda la columna está sin codificar y el
         contexto de todos sus bits es ContextZERO then
3          if Todos los bits de la columna son cero then
4              mqCoder.code(0, ContextRUNLENGTH, bitStream);
5          else
6              mqCoder.code(1, ContextRUNLENGTH, bitStream);
7              j ← índice del primer bit de la columna distinto de cero;
8              (b_0^j, b_1^j) ← representación de dos bits sin signo de j;
9              mqCoder.code(b_0^j, ContextUNIFORM, bitStream);
10             mqCoder.code(b_1^j, ContextUNIFORM, bitStream);
11             b_s ← bit de signo asociado a la posición j;
12             context ← contexto de signo de b_s;
13             x_s ← bit de XOR asociado a context;
14             mqCoder.code(b_s ⊕ x_s, context, bitStream);
15             Saltar hasta el bit en la posición j + 1 y volver a 2;
16         end
17     else
18         Codificar b como si fuera de significancia (Algoritmo 12);
19     end
20 end
```

- Tabla de cambio de bit. Señala para cada estado si es necesario cambiar el bit de predicción en caso de fallo para adaptarse mejor a la distribución de probabilidad cambiante de los bits del contexto asociado.
- Tabla de probabilidad. Asocia cada estado a un número que representa la probabilidad estimada de que la predicción sea certera. Solo tenemos 47 posibles probabilidades, una para cada estado.

Con todo esto, el comportamiento del codificador MQ puede observarse en el Algoritmo 15.

El cambio de la línea 6 es conocido como intercambio condicional, y mitiga la mayor pérdida de eficiencia en el codificador, que ocurre cuando la probabilidad es cercana a 1/2 y el símbolo más probable puede ser asignado el intervalo pequeño al partirlo añadiendo un nuevo símbolo. Con esto se asegura que el símbolo más probable mantiene el intervalo mayor.

Es en la renormalización de la línea 20 donde, al renormalizar A para que alcance los 16 bits de longitud se van generando nuevos bits en el registro C, que cada cierto tiempo serán emitidos, evitando por supuesto la formación de secuencias reservadas en el intervalo **0xff90-0xffff**.

5.9. Resultado de la compresión

El formato del archivo comprimido es propio, y su diseño busca utilizar la menor cantidad posible de memoria. Así, prácticamente la totalidad de los datos son binarios (con la salvedad de algunos metadatos que se guardan en texto plano, como vimos en la Sección 5.2.1.1). La estructura general puede verse en la Figura 5.9.

- **Número mágico**: Un único byte que indica el tipo de archivo: `0xff`.

Algoritmo 15: MQCoder.code

input: Un bit b y un contexto context
input: Una table que asocia contextos con estados
inout: Un bitStream donde verter los datos codificados

```
1  estado ← table [context ];
2  pred ← predicción asociada a estado;
3  prob ← probabilidad de la tabla de probabilidad asociada a estado;
4  this.A ← this.A − prob;
5  if prob cae fuera del nuevo subintervalo (this.A < prob) then
6      Cambiar pred al bit contrario, para en caso de acertar quedarnos el subintervalo
         grande;
7  end
8  if la predicción ajustada fue certera then
9      this.C ← this.C + prob nos quedamos con el subintervalo grande;
10 else
11     this.A ← prob nos quedamos con el subintervalo pequeño;
12 end
13 if es necesario renormalizar then
14     if acertamos en la predicción original sin ajuste then
15         Actualizar estado en favor de la predicción utilizando la transición de la tabla de
             símbolo más probable;
16     else
17         Actualizar estado en contra de la predicción con la tabla de símbolo menos
             probable, aplicando el cambio de bit si es necesario;
18     end
19 end
20 while es necesario renormalizar do
21     this.C ← this.C ∗ 2;
22     this.A ← this.A ∗ 2;
23     t̄ ← t̄ − 1;
24     if t̄ = 0 then
25         Formar un nuevo byte con la parte consolidada de C;
26         if podría formarse una secuencia reservada con el siguiente byte then
27             t̄ = 7, esperando 7 bits para formar un nuevo byte que no podrá formar
                 secuencia reservada, pues es de la forma 0b0xxxxxxx;
28         else
29             t̄ = 8, esperando 8 bits para formar un nuevo byte;
30         end
31     end
32 end
```

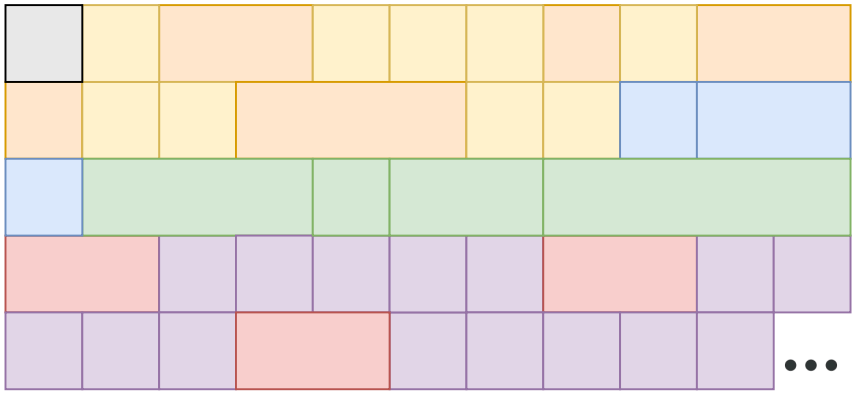


Figura 5.9: El número mágico es lo primero que aparece en el archivo comprimido. Seguido de los pares clave-valor. A continuación los diferentes metadatos de la compresión y de la reducción dimensional. Por último los metadatos de cada banda reducida seguidos de sus bloques codificados uno tras otro.

- **Metadatos de la imagen**: Pares clave-valor, o simplemente claves que indican datos como el tamaño de la imagen, formato de los datos, o dónde fue tomada.
- **Metadatos de la compresión**: Datos sobre el tipo de algoritmo utilizado para la reducción, número de bits a los que se ha comprimido, e incluso diferentes técnicas de compresión de la imagen.
- **Metadatos de la reducción dimensional**: Matriz de recuperación, número de dimensiones a las que se redujo, etc.
- Para cada banda:
  - **Cuantización**: Datos sobre los parámetros de cuantización, que cambian de banda a banda.
  - **Detección de puntos aislados**: Opcionalmente se pueden excluir puntos aislados de la compresión para mejorar la eficiencia.
  - **Función de precuantización**: Opcionalmente se puede aplicar una transformación invertible a los puntos antes de comprimirlos para cambiar su distribución y mejorar la compresión posterior.
  - Para cada bloque:
    - **Datos comprimidos**: Información comprimida de cada bloque.
    - **Marcador**: Marcador de terminación de bloque: `0xfffe`.

5.10. Decodificación

Durante las anteriores secciones hemos visto el proceso mediante el cual se iba transformando la imagen a través de las diferentes etapas, hasta que acababa comprimida del todo. Es necesario además poder realizar el proceso inverso, recuperando una aproximación de la imagen original más parecida cuanta menos compresión apliquemos.

Para ello se invierte bloque a bloque el proceso realizado por los codificadores MQ y EB. Una vez obtenidos los bloques se monta la imagen reducida. A continuación se descuantiza y se aplica la ondícula inversa, para finalmente deshacer la reducción de dimensionalidad y volver al tamaño original.

Lectura de metadatos:

En primer lugar necesitamos saber metadatos de la imagen como tamaño, tipo de datos, algoritmo utilizado, etc. El formato de guardado ya lo hemos comentado en la Sección 5.2.1.1. Al estar definido el formato por nosotros mismos, no tendremos más que leer el byte que nos indica la clave, y a continuación sabremos el tipo del valor que prosigue, que podremos leer de manera binaria sin problemas. Una vez cargados todos los metadatos, podemos pasar al siguiente paso:

Decodificador MQ y EB:

Como vemos en el Algoritmo 16, el proceso de decodificación es muy similar al de codificación, cambiando algunos pasos de orden para obtener los bits codificados del flujo de entrada.

Los contextos, generados por el EBCoder, se generan de la misma manera. Al depender solo de muestras previas, en la decodificación ya disponemos de ellas, por lo que es trivial reconstruir el mismo contexto que se utilizó para la codificación. Es decir, el proceso de decodificación es el mismo que el de codificación presentado en el Algoritmo 11, con la salvedad de los parámetros de entrada y salida.

Descuantización:

El proceso de descuantizado también es trivial. Aquí es donde se pierde la precisión ya que la cantidad redondeada en el proceso de cuantización se pierde por el camino. Así pues invertimos el valor redondeado con la fórmula inversa a la de cuantización, obteniendo una aproximación del valor original.

Ondícula inversa:

La ondícula inversa simplemente utiliza unos kernels diferentes a los de la transformación original. La aplicación se hace exactamente de la misma manera, por lo que no tiene mayor dificultad.

Aumento dimensional:

Si recordamos lo visto en la Sección 5.3, las funciones de entrenamiento generaban tanto los datos necesarios para la reducción (W, $\bar{\boldsymbol{x}}$, etc), como los necesarios para el aumento ($\bar{W}$, etc). Por tanto para recuperar la dimensión original de la imagen, tan solo tenemos que leer del archivo comprimido los datos del algoritmo utilizado y aplicar la función `boost` ya definida.

Algoritmo 16: MQCoder.decode

input : Un contexto context
input : Una table que asocia contextos con estados
inout : Un bitStream de donde se van extrayendo bits codificados
output: El bit b decodificado

```
1  estado ← table [context ];
2  pred ← predicción asociada a estado;
3  prob ← probabilidad asociada a estado;
4  this.A ← this.A − prob;
5  if this.A < prob then
6      Cambiar pred al bit contrario;
7  end
8  if el intervalo no coincide con el esperado (this.C < prob) then
9      b ← inverso de pred;
10     this.A ← prob actualizando al subintervalo pequeño;
11 else
12     b ← pred;
13     this.C ← this.C + prob cambiando al intervalo grande;
14 end
15 if es necesario renormalizar then
16     if el símbolo recuperado coincide con la predicción then
17         Actualizar estado en favor de la predicción;
18     else
19         Actualizar estado en contra de la predicción;
20     end
21 end
22 while es necesario renormalizar do
23     if t̄ = 0 then
24         Rellenar C con el siguiente byte leído;
25         if se formó una secuencia reservada con el byte leído then
26             t̄ = 7, solo tenemos 7 bits útiles nuevos;
27         else
28             t̄ = 8, los 8 bits inyectados son válidos;
29         end
30     end
31     this.C ← this.C ∗ 2;
32     this.A ← this.A ∗ 2;
33     t̄ ← t̄ − 1;
34 end
```

Capítulo 6

Implementación y rendimiento

6.1. Jypec

El algoritmo desarrollado en el Capítulo 5 se ha implementado en Java utilizando el IDE Eclipse [68]. El código fuente se encuentra en GitHub bajo el nombre JYPEC [69] (*Java hYPErspectral Compressor*). A rasgos generales, la implementación sigue el diseño orientado a objetos de Java, con un diseño de clases intercambiables que hacen que el algoritmo sea altamente personalizable. Destacamos en esta sección algunos diagramas de clases[1] concretos que resultan interesantes para conocer cuál es la implementación del flujo general del programa.

6.1.1. Flujo general del programa

El programa cuenta con cuatro funciones básicas: Comprimir, descomprimir, analizar una imagen y realizar la comparativa entre dos imágenes. La Figura 6.1 muestra el flujo del programa.

Como podemos ver, tras decidir la operación a realizar, se pasa a leer la imagen. La lectura incluye de por sí la compresión, así que podemos recomprimir una imagen, o comparar directamente una imagen comprimida con otra que no lo esté.

- **Comprimir**: La imagen se comprime según los argumentos especificados en la Sección 6.2, siguiendo el algoritmo del Capítulo 5.
- **Descomprimir**: Proceso inverso a la compresión. Los argumentos especificados no tienen efecto ya que se deben utilizar los mismos que se utilizaron en la compresión, los cuales se han guardado en los metadatos de la imagen comprimida.
- **Comparar**: Se leen y comparan dos imágenes distintas, mostrando las medidas de diferencia especificadas en la Sección 6.3.
- **Analizar**: Muestra estadísticas sobre el número de píxeles cuyo valor se desvía excesivamente de una interpolación de sus vecinos. Útil para detectar si una imagen contiene ruido de sal y pimienta[2] por errores en el sensor o el proceso de compresión/descompresión.

6.1.2. Estructura de la imagen en memoria

En primer lugar tenemos la estructura de la imagen hiperespectral en la Figura 6.2.

[1] En verde se señalan las clases normales, en naranja los enumerados, en azul las clases abstractas, y en morado las interfaces. Los paquetes tienen su propio símbolo, y algunas clases concretas que sólo tienen constantes para configuración se marcan en rojo.

[2] Píxeles blancos y/o negros esparcidos por la imagen.

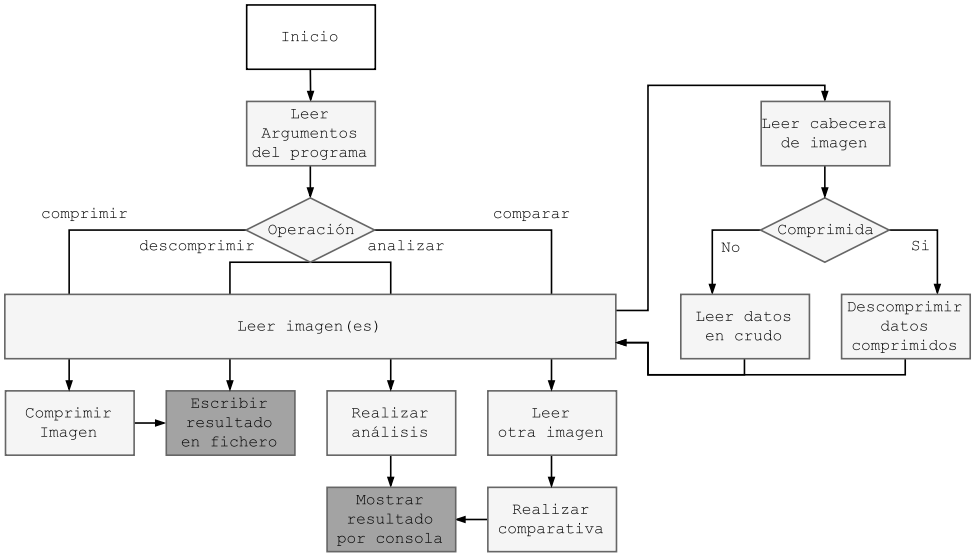


Figura 6.1: Flujo seguido por Jypec en la ejecución. Observamos las diferentes funcionalidades.

La imagen consta de dos partes bien diferenciadas: la cabecera y los datos. El objeto de cabecera mapea un número arbitrario de campos a objetos que contienen datos asociados a los mismos. Los datos, por su parte, pueden ser de diversos tipos (entero sin signo de un byte, con signo de dos bytes...). Independientemente del tipo de datos, los valores pueden guardarse en diferentes contenedores, que en nuestro caso son bien enteros o bien coma flotante. Dependiendo del tipo, el contenedor podrá provocar pérdidas de información (redondeos, truncamientos...) por lo que hay que asegurarse de utilizar el adecuado para nuestro propósito.

Por último, el tipo entero permite la instanciación de bandas individuales asociadas, a fin de facilitar el proceso de codificado mediante operaciones de creación de bloques.

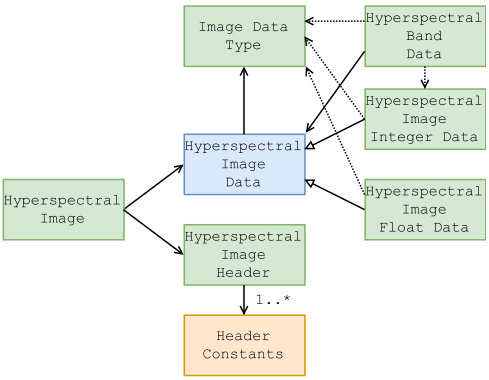


Figura 6.2: Diagrama de clases que muestra las estructuras de datos que alberga una imagen hiperespectral.

6.1.3. Lectura y escritura de los metadatos

Lo primero que hacer al tratar con una imagen es procesar los metadatos. Los metadatos son una lista de *parámetros* (parejas clave-valor). En la lectura los necesitamos para saber si la imagen está o no comprimida, y conocer el tipo de datos que vamos a encontrar. En la escritura es lo primero que debe escribirse, para poder realizar posteriormente el proceso de lectura sin problemas.

Como la funcionalidad principal del programa es la de comprimir y descomprimir, los metadatos que se lean serán los mismos que se escriban. Así, la lectura y escritura de las cabeceras está totalmente acoplada, como vemos en la Figura 6.3. Todas las clases del diagrama incluyen métodos para la lectura y escritura, tanto de su forma comprimida como descomprimida.

`ImageHeaderReaderWriter` es la clase base que contiene todos los métodos para lectura y escritura. Las demás contienen métodos análogos, que serán invocados según sea necesario.

Lectura: En primer lugar se consulta el número mágico del archivo, que será `0xff` para archivos comprimidos (Sección 5.9), y `'E'=0x45` para los no comprimidos (Sección 5.2).

Cabecera comprimida: Se leen las parejas clave-valor. La clave siempre es un byte, y la longitud del valor (conocida de antemano) dependerá de la clave. Una vez leídas ambas se guardan en un parámetro.

Cabecera no comprimida: Utilizando expresiones regulares se separa la cabecera en las parejas clave-valor, que se parsean según el tipo de la clave, y son guardadas también en parámetros.

Escritura: En primer lugar se guarda el número mágico. Posteriormente cada parámetro, que tiene dos formas de guardado: con y sin comprimir.

Cabecera comprimida: Para cada parámetro se escribe un byte indicando su tipo, y a continuación el valor, en un formato previamente especificado, para que ocupe siempre lo mismo.

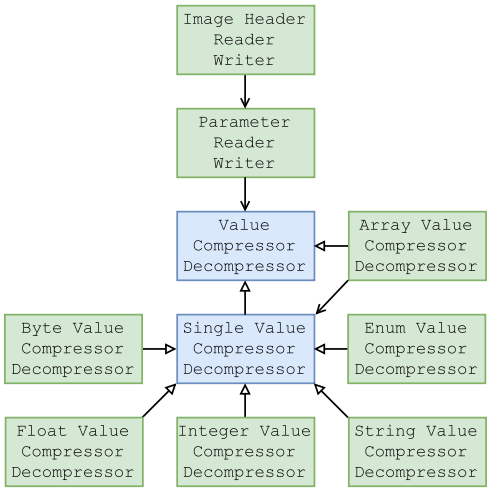


Figura 6.3: Diagrama de clases que muestra la jerarquía existente para cargar y escribir las cabeceras de imágenes.

Cabecera no comprimida: Se escriben, en texto plano, las parejas clave-valor.

6.1.4. Lectura y escritura de datos planos

Si las imágenes no están comprimidas, los datos referentes a todas las muestras se guardan en crudo, uno tras otro, siguiendo un formato determinado impuesto por los metadatos (tipo de datos, profundidad de bits, ordenación de las muestras, número de muestras, etc).

Tanto la lectura como la escritura se hacen de manera similar. El lector/escritor manda los metadatos a una factoría, que devuelve un lector/escritor concreto para el tipo de datos requerido. Los datos serán leídos/escritos del/al archivo especificado. Esta simetría puede observarse en la Figura 6.4

Vemos además cómo el lector/escritor de la imagen hiperespectral completa tienen acceso tanto a los datos como a los metadatos, utilizando estos últimos para manipular los primeros.

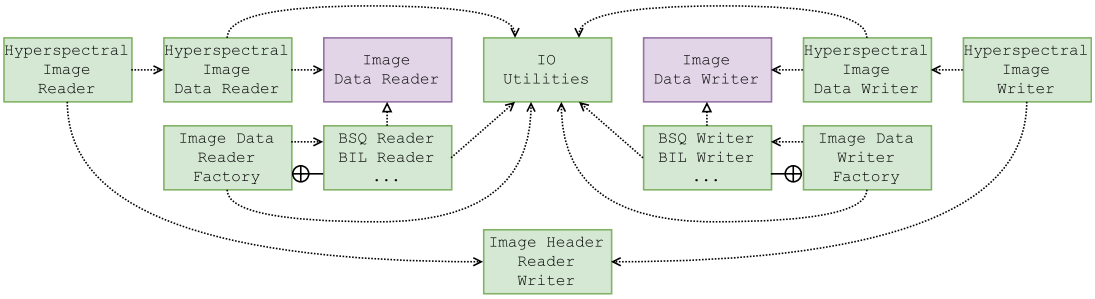


Figura 6.4: Diagrama de clases que muestra la jerarquía existente para la entrada y salida de imágenes completas, y en concreto los datos en crudo.

6.1.5. Compresión / Descompresión

Vista la manipulación de los datos en crudo, pasamos a detallar qué clases principales (Figura 6.5) intervienen en la compresión y descompresión de los datos.

Lo más destacable es observar la simetría existente entre la compresión y la descompresión. Quitando el codificador de bloque y los flujos de entrada y salida, el resto de clases tienen todas los métodos necesarios para hacer (y deshacer) su efecto sobre la imagen para comprimir o descomprimir respectivamente.

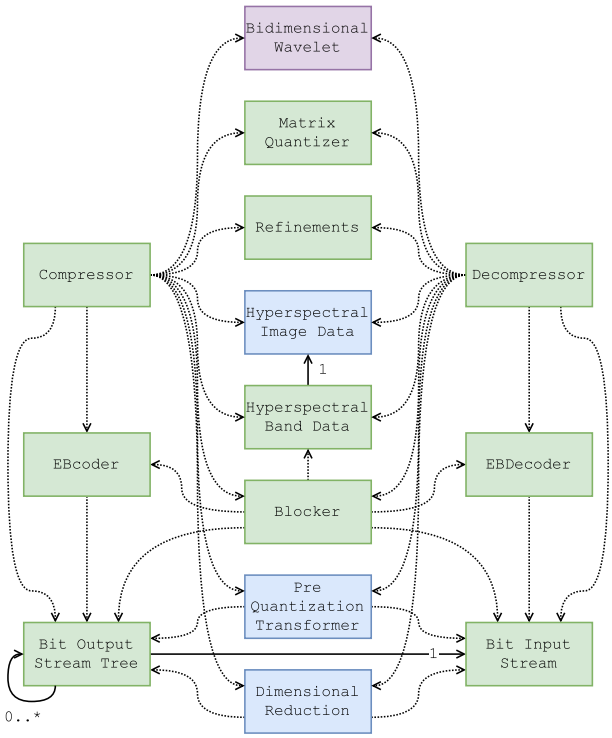


Figura 6.5: Diagrama de clases principales que participan en el proceso de la compresión y descompresión. Obsérvese la simetría.

Ambos procesos van instanciando las diversas clases que forman parte de la compresión según son necesarias. Algunas solo aplican transformaciones a los datos tras ser configuradas con datos de la cabecera, y otras además de transformar los datos interactúan directamente con los flujos de entrada o salida, leyendo o escribiendo los datos que sean pertinentes.

En cuanto a los flujos de entrada y salida, son ligeramente diferentes. El de salida es capaz de tener subflujos dentro del mismo, a fin de diseñar una suerte de "árbol" para poder ver cuánto ocupa cada parte del archivo comprimido y depurar mejor errores o detectar las zonas menos comprimidas. También es capaz de funcionar como un flujo normal de búfer único. El flujo de entrada es simplemente un envoltorio que admite operaciones bit a bit sobre el flujo subyacente. Para evitar la necesidad de volcar a archivo para hacer pruebas de descompresión, el flujo de salida puede proporcionar directamente un flujo de entrada que bebe de los bits escritos en el primero.

Destacamos un par de diagramas dentro de la compresión: el de la reducción dimensional y el de la codificación de los datos.

6.1.5.1. Reducción dimensional

Observamos la estructura del paquete de la reducción dimensional en la Figura 6.6.

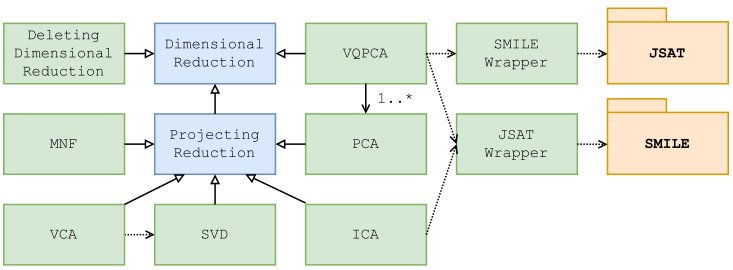


Figura 6.6: Diagrama de clases que muestra la implementación de los diferentes algoritmos de reducción dimensional. JSAT y SMILE son bibliotecas externas.

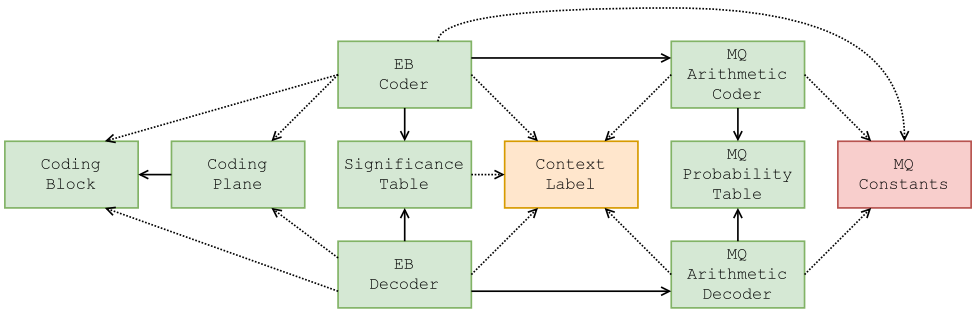


Figura 6.7: Diagrama de clases referente a los codificadores de bloque y aritmético MQ.

Aquí se ve claramente qué algoritmos tienen dependencias externas y cuáles no. La estructura de clases parte de la clase base de reducción dimensional, y se especializa con una segunda de reducción mediante proyección (al que pertenecen todos los algoritmos excepto VQPCA y el de borrado de dimensiones (utilizado en tests)).

Observamos también el uso, por parte de ICA y VQPCA, de las bibliotecas externas JSAT [58] y [62], gracias a sus respectivos wrappers para la API.

6.1.5.2. Codificación de los datos reducidos

También tenemos, por supuesto, el diagrama de las interacciones entre el EBCoder y el MQCoder (Figura 6.7), cuyos algoritmos veíamos en el Capítulo 5.

Las clases que hacen de interfaz para los datos son el bloque y plano de codificación, tanto para el EBCoder como para el EBDecoder. Ambos llevan cuenta de la significancia mediante la clase tabla de significancia, pues siempre está sincronizada entre ambos módulos. Además, tanto codificador como decodificador de bloque cuentan con sus respectivos homólogos aritméticos. Estos van también sincronizados mediante su propia tabla de probabilidades y estados, y se configuran mediante un archivo de constantes.

6.2. Opciones del programa

Jypec ofrece una variada selección de opciones. Sus cuatro funciones principales (comprimir, descomprimir, analizar y comparar) van acompañadas de diferentes modificadores. A continuación se listan todas las opciones disponibles mediante línea de comandos.

`-a --analyze`

Lanza la utilidad de análisis de imagen. Básicamente detecta si hay píxeles que difieran mucho de sus vecinos. En ocasiones las imágenes hiperespectrales tienen píxeles "muertos" sin información. Esto afecta muy negativamente a la compresión, que espera cierta redundancia espacial. Una imagen que en el análisis detecte muchos píxeles dispares debería ser reparada antes de comprimir.

`-b --bits <quantity>`

Número de bits que utilizar para la codificación. Al menos 2 (signo-magnitud).

`-c --compress`

Comprime la imagen de entrada.

`-d --decompress`

Descomprime la imagen de entrada.

--essential_header

Solo empaqueta la información imprescindible para la descompresión de la imagen, eliminando de la cabecera comprimida todos los demás metadatos.

-h --help

Muestra un mensaje de ayuda con información sobre los comandos, bastante más breve que esta sección pero útil para emergencias.

-h --hardcode_outliers <percentage>

Especifica el porcentaje (número en coma flotante entre 0 y 1) de las muestras que se codificarán en crudo antes de aplicar el EBCoder. Dichas muestras se tomarán de los extremos, reduciendo el intervalo a codificar (ver Sección 5.4).

-i --input <file>

Fichero de entrada. No importa que esté o no comprimido. En cualquier caso la aplicación lo abrirá si dispone de la cabecera adecuada para hacerlo.

--input_header <file>

Fichero opcional de cabecera. Si no se da, se asume que la cabecera es lo primero que hay en el fichero de entrada, y a continuación siguen los datos.

-k --compare

Además de leer el fichero de entrada, se lee el especificado como salida, y se computan diferentes métricas de diferencia entre los mismos. Esto sólo se hace si ambas imágenes coinciden en dimensiones y tipo de datos.

--no_header_output

No escribir cabecera para el fichero de salida. Esta opción solo es admitida en la descompresión. En la compresión es imprescindible empotrar la cabecera en el archivo de salida. No obstante, se puede reducir a su mínima expresión con la opción **--essential_header**.

-o --output <file>

Fichero de salida, o archivo a comparar con el de entrada si se utiliza la opción **--compare**.

--output_header <file>

Nombre del fichero de salida de cabecera.

--prequantize (log | sqrt | split | none)

Opción para activar la precuantización (Sección 5.6.1) con la función dada, a elegir entre las cuatro opciones disponibles. Por defecto no se aplica precuantización.

-r --reduction <algorithm>

Especifica el algoritmo de reducción a utilizar. Cada uno tiene sus peculiaridades:

- (pca | mnf | ica) **<components>**

  Especifica los métodos de PCA, MNF o ICA, con el número de componentes que se desean preservar.

- svd **<components>** [** **]

  Análogo al anterior pero para SVD, con la opción de centrar o no los datos previa transformación.

- vqpca **<components> <clusters>** [(SMILE | JSAT)]

Utiliza el método VQPCA con los components y clusters indicados, y opcionalmente se puede especificar la biblioteca a utilizar.

- `vca components [<seed>]`

  Además de la opción de componentes, se puede especificar la semilla para asegurar el mismo resultado en ejecuciones diferentes. De otra manera resultados para los mismos parámetros podrían ser diferentes.

`-s --shave` `[(<band> <quantity>)...]`

Podemos especificar los bits que se rebajarán del valor por defecto especificado con **`-b`**. Se especifican mediante una lista de números. El parseador los toma de dos en dos, especificando el primero la banda donde se hace la reducción, y el segundo la cantidad de bits que se rebajan. Adicionalmente, todas las bandas posteriores a la indicada tendrán esa reducción. Si se desea rebajar por ejemplo solo en la banda 2, habría que especificar: **`-s`** `2` **`<quantity>`** `3 0 .`

`--stats`

Muestra estadísticas de profiling de la ejecución, devolviendo el tiempo empleado en cada una de las funciones medidas activamente, así como los porcentajes de cada una sobre el total.

`-t --train_reduction <percentage>`

Porcentaje de muestras con las que se quiere entrenar al algoritmo especificado con **`-r`**. Se realizará un submuestreo de la entrada para el entrenamiento, acelerando así el proceso. Posteriormente se aplicará el conocimiento adquirido con ese entrenamiento para reducir la totalidad de las muestras.

`--tree`

Genera un árbol representativo del archivo comprimido, con su mismo nombre acabado en `.tree`. En él se puede ver la estructura interna del archivo, y analizar cuánto ocupa cada parte del mismo, para ver dónde existe margen de mejora, o si algo está ocupando más de lo que debería.

`-v --verbose`

Se va mostrando información en tiempo real mientras se ejecutan los algoritmos. Útil para saber por dónde va la ejecución si esta se demora. Junto a cada salida, se muestra la posición de código que la generó, además del momento en el tiempo en que se pasó por la misma.

`-w --wavelet <passes>`

Especifica el número de pasadas de transformada de ondícula que se deben aplicar.

6.3. Métricas de comparación

6.3.1. Calidad

Una parte muy importante al hacer compresión con pérdida es medir la distorsión de la señal comprimida respecto a la original. Las diferentes métricas de distorsión devuelven un único valor que indica la calidad de la imagen reconstruida. Resumir tanta información en un único número es bastante ambicioso, pues la similitud es muchas veces relativa. Por eso se han incorporado varias métricas en el programa, cada una centrándose en ciertos aspectos, a fin de tener más información sobre la calidad de la reproducción.

Definición 8 *El **error cuadrático máximo** (maxSE) se define como el máximo de los cuadrados de las diferencias entre todas las muestras de dos imágenes. Denotando una imagen I como una sucesión ordenada de muestras $I = \{i_1, \ldots, i_n\}$:*

$$maxSE(I^a, I^b) = \max_{j=1}^{n}(i_j^a - i_j^b)^2 \tag{6.1}$$

Esta métrica es útil para detectar la distorsión máxima en una muestra de la imagen. Si el maxSE es pequeño, podemos estar seguros de que la distorsión general será pequeña también. Sin embargo, pequeñas anomalías pueden disparar el maxSE sin ser importantes para la distorsión general de la imagen, por lo que se suelen utilizar otras métricas diferentes.

Definición 9 *El **error cuadrático medio** (MSE) se define como la media de los cuadrados de las diferencias entre todas las muestras de dos imágenes.*

$$MSE(I^a, I^b) = \frac{\sum_{j=1}^{n}(i_j^a - i_j^b)^2}{n} \tag{6.2}$$

Esta medida nos da una idea de la cantidad absoluta que se desvían nuestras muestras de las originales, pero puede ser muy engañosa: Imaginemos que las muestras de nuestra imagen original son de valor muy pequeño (por facilidad de ejemplo todas valen 1), y en la imagen comprimida solo aparecen los valores 0 y 2. Las muestras se están anulando completamente o incluso duplicando su valor, sin embargo el MSE será menor que uno. Y a la inversa pasa algo similar, si los valores de nuestras muestras son muy altos, un pequeño porcentaje de desviación hará que el valor absoluto de desviación sea muy grande, y por tanto también lo sea el MSE.

Definición 10 *El **ratio señal ruido pico** (PSNR) se define en función del MSE y del rango dinámico de la imagen $r_{\text{máx}}(I)$ como:*

$$PSNR(I^a, I^b) = 10\log_{10}\left(\frac{r_{\text{máx}}(I^a)^2}{MSE(I^a, I^b)}\right) \tag{6.3}$$

Aquí tomamos como referencia el valor máximo que pueden tomar las muestras, y evaluamos la distorsión medida con MSE relativa al mismo. Esta medida da un valor fiable cuando las muestras tienen valores no muy alejados del máximo. Al igual que ocurría con el MSE, si las imágenes tienen valores muy pequeños respecto al rango dinámico, el PSNR será muy bueno pese a que la imagen pueda ser irreconocible. No obstante, PSNR nos sirve para detectar imágenes notablemente malas. Si falla esta métrica, probablemente fallen las demás.

Definición 11 *El **ratio señal ruido pico normalizado** (NPSNR) sigue la idea del PSNR, pero en lugar de tomar como referencia el valor máximo que puede tomar una muestra, toma el rango dinámico de la imagen:*

$$NPSNR(I^a, I^b) = 10\log_{10}\left(\frac{(\text{máx}(I^a) - \text{mín}(I^a))^2}{MSE(I^a, I^b)}\right) \tag{6.4}$$

La ventaja de esta definición es que ahora ya no tenemos el problema de que imágenes con muestras de valor pequeño nos puedan dar resultados de distorsión mejores que la realidad, y sigue teniendo la ventaja de que un NPSNR malo probablemente implique que el resto de medidas son malas. Además, al contrario que PSNR, el NPSNR es invariante ante cambios de

rango dinámico de la imagen (límites inferior y superior de valores en las muestras) siempre y cuando no se tengan que truncar las mismas en el cambio. Siempre se cumple que:

$$NPSNR(I^a, I^b) \leq PSNR(I^a, I^b)$$

Definición 12 *El **ratio señal ruido normalizado con potencia** (POWSNR) se define como:*

$$POWSNR(I^a, I^b) = 10 \log_{10} \left(\frac{pow(I^a)}{MSE(I^a, I^b)} \right) \tag{6.5}$$

*Donde $pow(I)$ es la **potencia** de la imagen I, definida como:*

$$pow(I) = \frac{\sum_{j=1}^{n} (i_j)^2}{n}$$

Esta métrica es menos sensible a la existencia de valores extremos que NPSNR, y por tanto más fiable si nuestra imagen tiene algo de ruido. Esta métrica siempre cumple:

$$POWSNR(I^a, I^b) \leq NPSNR(I^a, I^b)$$

Definición 13 *El **ratio señal ruido** (SNR) se define como:*

$$SNR(I^a, I^b) = 10 \log_{10} \left(\frac{\sigma(I^a)^2}{MSE(I^a, I^b)} \right) \tag{6.6}$$

Donde $\sigma(I)^2$ es la varianza de I:

$$\sigma(I)^2 = \frac{\sum_{j=1}^{n} (i_j - \mu(I))^2}{n}$$

Siendo $\mu(I)$ el valor medio de las muestras de I.

Esta definición tiene en cuenta, hasta cierto punto, la estructura de la imagen. Al introducir la varianza en la ecuación, se es más permisivo con la distorsión en imágenes con muchos cambios, exigiendo que las imágenes más planas tengan menos distorsión. Para imágenes normales, un SNR de 32 se considera [70] calidad excelente, siendo hasta 20 un SNR de calidad aceptable.

Esta métrica también se abstrae del valor absoluto de las muestras, y cumple que:

$$SNR(I^a, I^b) \leq POWSNR(I^a, I^b)$$

Es importante destacar, llegados a este punto, que de las cuatro métricas referidas al ratio señal ruido, solo PSNR es simétrica:

$$PSNR(I^a, I^b) = PSNR(I^b, I^a)$$

El resto de métricas no cumplen la igualdad, y por tanto debemos asegurarnos de tomar siempre la misma imagen de referencia para tener unos resultados consistentes.

Definición 14 *El **ratio de media a desviación estándar** (MSR) se define como el cociente entre la media $\mu(I)$ de los valores de una imagen referencia respecto a la desviación estándar entre las dos imágenes a comparar:*

$$MSR(I^a, I^b) = \frac{\mu_(I^a)}{\sqrt{MSE(I^a, I^b)}} \tag{6.7}$$

Nos da una buena aproximación de la cantidad de error que podemos esperar en una muestra. Un valor x indica que por cada x unidades de muestra, podemos esperar una desviación de 1. (Por ejemplo, si $MSR = 100$ una muestra que valga 1200 probablemente se habrá distorsionado al rango $(1188, 1212)$).

Definición 15 *El **índice de similitud estructural** [71] (SSIM) se define como:*

$$SSIM(I^a, I^b) = \frac{\left(2\mu(I^a)\mu(I^b) + c_1\right)(2\sigma + c_2)}{\left(\mu(I^a)^2 + \mu(I^b)^2 + c_1\right)\left(\sigma(I^a)^2 + \sigma(I^b)^2 + c_2\right)} \tag{6.8}$$

Donde σ es la covarianza de las dos imágenes I^a e I^b, y c_1 y c_2 son factores de corrección, definidos como:

$$c_1 = k_1^2 r_{\text{máx}}^2 \qquad c_2 = k_2^2 r_{\text{máx}}^2$$

Con k_1 y k_2 a elegir, usualmente 0,01 *y* 0,03 *respectivamente.*

Esta métrica ofrece un valor decimal entre 0 y 1, que indica mayor similitud cuanto más cerca de 1 se está. Si bien está más enfocada a la similitud detectaba por el ojo humano, también nos sirve para hacernos una muy buena idea de cuánta información se puede estar perdiendo de cara a hacer un análisis numérico posterior de la imagen.

6.3.2. Compresión

En la compresión solo estamos interesados en cuál es el tamaño comprimido respecto al original. Pese a la sencillez de lo que buscamos medir, hay diferentes formas de expresarlo.

Una primera posible métrica son los *bpppb* (*Bits per pixel per band*). Esto indica los bits utilizados para representar cada muestra individual de la imagen. Por ejemplo, si una imagen está comprimida a $0{,}1bpppb$, y tiene un tamaño de $100 \times 100 \times 100$, en total ocupará $100^3 \cdot 0{,}1 = 100000b$. Si la imagen original ocupaba $16Mb$, la habremos comprimido 160 veces. El problema de esta métrica es que es necesario saber cuántos bits se utilizaban por muestra previa compresión. $0{,}1bpppb$ representa una compresión de 160 veces en caso de tener la imagen una profundidad de $16b$, y una compresión de 80 si la profundidad era $8b$. Además, tampoco queda claro si la compresión se refiere exclusivamente a los datos de la imagen o incluye los metadatos.

Para evitar esa confusión, aquí se ha decidido utilizar el *ratio* o *tasa* de compresión (utilizados indistintamente con el mismo significado). El ratio de compresión indica en cuántas veces se ha dividido el tamaño original para obtener la imagen comprimida, incluyendo todos los metadatos necesarios para recuperar la original. Por definición no tiene unidades, es simplemente un número:

$$\text{ratio} = \frac{\text{tamaño en bits sin comprimir}}{\text{tamaño en bits comprimido}}$$

Imagen	N_z	N_x	N_y	Tamaño	Tipo de datos	Orden	Endianness
SUW	360	320	1200	276,48MB	`uint16`	BIL	Little
DHO	360	320	1160	267,264MB	`uint16`	BIL	Little
BEL	360	320	600	138,24MB	`uint16`	BIL	Little
REN	356	320	600	136,704MB	`uint16`	BIL	Little
CRW	224	614	512	140,836MB	`int16`	BIP	Big
CUP	188	350	350	46,06MB	`int16`	BSQ	Little

Tabla 6.1: Metadatos más importantes de las diferentes imágenes de prueba. $N_z \times N_x \times N_y = N_{bands} \times N_{samples} \times N_{lines}$.

Un ratio de compresión $r = 160$ indica que la imagen comprimida ocupa 160 veces menos que la original. Siguiendo el ejemplo anterior, sería equivalente a 0,1*bpppb* en caso de tener 16 bits de profundidad. Si la profundidad fuera de 8 bits, un ratio de 160 equivaldría a 0,05*bpppb*.

6.4. Imágenes de prueba

Para las pruebas se han seleccionado un total de 6 imágenes [72], con diferentes tamaños y características. Para todas ellas se utilizarán, en lo que resta de documento, las abreviaturas que se indican a continuación. Podemos ver más detalles de las imágenes en la Tabla 6.1.

- **SUW**: Muestra de pantano del golfo de Méjico, en la reserva natural "Lower Suwannee". $360 \times 320 \times 1200$.
- **DHO**: Imagen del océano tras la fuga de crudo del "Deepwater Horizon". $360 \times 320 \times 1200$.
- **BEL**: Zona de cultivos rodeada por bosque en Beltsville, MD, USA. $360 \times 320 \times 600$.
- **REN**: Zona mixta urbana y salvaje cerca en Reno, NV, USA. $356 \times 320 \times 600$.
- **CRW**: Imagen completa del sensor AVIRIS del valle de Cuprite, NV, USA sin corrección espacial. $224 \times 614 \times 512$.
- **CUP**: Imagen corregida y recortada del sensor AVIRIS, también tomada en el valle de Cuprite. $188 \times 350 \times 350$.

Una imagen hiperespectral está formada por datos tridimensionales. Debido a las limitaciones de representación en un formato bidimensional, a lo largo de este trabajo se tomará una representación de la imagen donde se han escogido las bandas 29, 19 y 9 como representantes de los colores rojo, verde y azul respectivamente. En ocasiones también se han realizado correcciones de brillo y contraste para distinguir mejor las características. Podemos ver algunos ejemplos en la Figura 6.8.

6.5. Ejecutando el algoritmo

Ya tenemos el algoritmo, las imágenes, y la forma de medir la distorsión entre la versión original y comprimida. A lo largo de las siguientes páginas iremos listando comandos junto con los resultados de compresión que ofrecen. Por simplicidad se lista solo el comando de compresión, omitiendo las opciones de entrada, salida y detalles de ejecución. Sólo nos centramos en los parámetros del algoritmo.

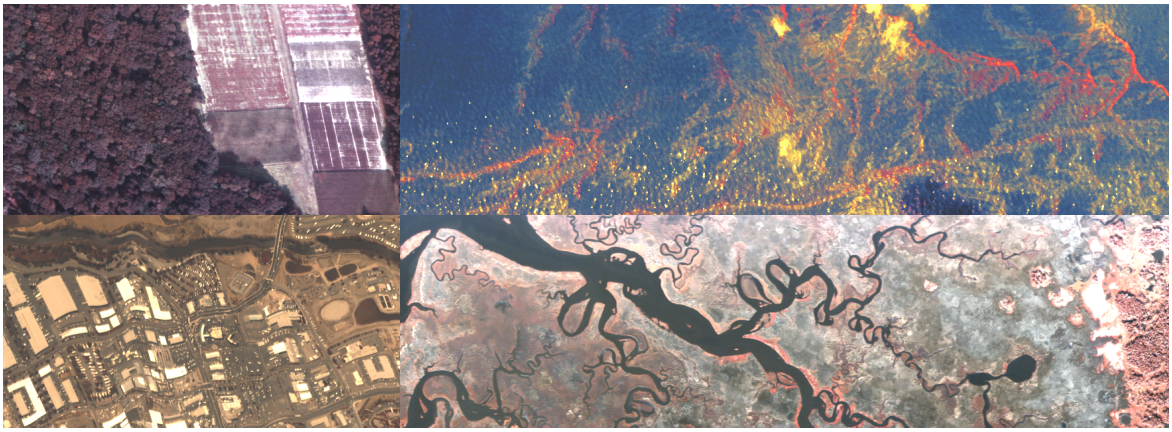


Figura 6.8: Arriba: BEL y DHO. Abajo: REN y SUW. Observamos como REN no ha sido corregida espacialmente, y presenta artefactos curvos. No será problema para la compresión. Nota: Los colores no tienen por qué corresponderse con la realidad.

Imagen	Tiempo (s)	Ratio	SNR	SSIM
SUW	07.581	486.84	30.04	0.99984
DHO	07.513	1914.37	25.39	0.99968
BEL	07.654	348.88	30.85	0.99966
REN	07.391	399.37	27.28	0.99910
CRW	09.209	121.25	21.28	0.99611
CUP	04.333	185.81	23.19	0.99921

Tabla 6.2: Resultados de la compresión de las diferentes imágenes para las opciones `-c -w 3 -b 10 -r pca 4 -t 0.01`

Para comenzar, vamos a ejecutar la compresión sobre las imágenes con los mismos parámetros. La elección no es arbitraria, sino que viene motivada por resultados que se verán a continuación. En cualquier caso, es interesante ver esto primero para mostrar una peculiaridad del algoritmo.

`-c -w 3 -b 10 -r pca 4 -t 0.01`

Lo primero que observamos mirando la Tabla 6.2 es la disparidad entre los resultados. Pese a comprimirse con los mismos parámetros, los ratios de compresión varían desde unas 100 veces más pequeño que el archivo original, hasta casi 2000. Vemos también que la imagen menos comprimida es la que peor calidad preserva, mientras que otras mucho más comprimidas tienen ratios bastante mayores. El tiempo tampoco se ajusta al tamaño de la imagen. Sí es verdad que la más pequeña es la que se comprime más rápido, pero las más grandes no son las que más han tardado.

Con todo esto, lo primero que queremos evidenciar es que cada imagen es un mundo, y no se pueden encontrar unos parámetros de compresión que funcionen a la perfección para todas. Sí existirán comportamientos más o menos generales, y ciertos parámetros afectarán de manera parecida a todas las imágenes. Intentaremos descubrir estas tendencias para recomendar unos parámetros de partida que deberán ser ajustados a cada caso particular.

Además, observamos que entre las dos medidas más prometedoras de calidad, SNR y SSIM, vamos a detectar mejor las diferencias en calidad con SNR. Para las calidades en las que nos moveremos ($> 15dB\ SNR$) la medida SSIM nos está dando siempre valores muy similares pues las imágenes son suficientemente buenas. Así que, para evitar tener que utilizar varios decimales, emplearemos SNR a lo largo de los resultados, con valores mayores indicando mejor calidad.

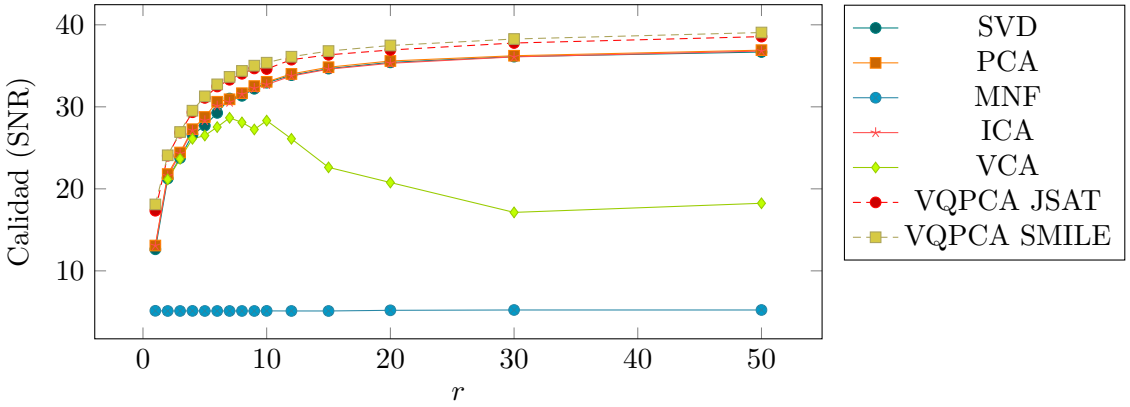


Figura 6.9: Calidad de imagen según aumenta r para los diferentes algoritmos implementados. (Compresión de REN).

6.5.1. Probando los reductores dimensionales

A continuación vamos a probar los diferentes reductores dimensionales, cambiando r para ver el efecto de la bajada de dimensión. Los algoritmos se han ejecutado con los parámetros por defecto, con un par de detalles para los que requieren más opciones:

- Para SVD no se ha aplicado el centrado de los datos.
- Para VQPCA se ha utilizado $t = 1$ en todas las ejecuciones, pues es el único algoritmo sensible a cambios agresivos en dicha variable. Además se ha decidido que el número de clusters sea 5.

Lo primero que observamos en la Figura 6.9 es que casi todos los algoritmos siguen la misma curva conforme aumenta r. Hay dos excepciones: MNF y VCA.

En el caso de MNF, la señal es distorsionada "a propósito" pues se realiza una estimación del ruido, que es eliminado. La comparación se sigue haciendo con la imagen original con ruido, de ahí los resultados tan anómalos. En cualquier caso, es claro ver que, a menos que pudiéramos partir del ruido real, este algoritmo no es útil utilizando estimaciones ya que la imagen resultante dista demasiado de la original.

Por parte de VCA, observamos que tiene un crecimiento en línea con las demás, hasta que se descuelga y empieza a perder calidad. VCA estima un recinto sobre las muestras, las cuales expresa como combinación lineal de sus vértices. Una vez la dimensión requerida supera la real (aproximadamente donde los demás algoritmos dejan de mejorar), VCA estima más vértices de los necesarios, y pierde rendimiento al expresar las combinaciones lineales en función de vértices que no aportan información.

Miramos ahora los tiempos de ejecución en la Figura 6.10. Vemos como VQPCA es con diferencia el que más tarda (especialmente con la biblioteca SMILE), pero los demás van más o menos parejos. Eso sí, PCA es el que mejor rendimiento conserva en valores altos de r, despegándose de todos los demás.

Por último comparamos la tasa de compresión obtenida en la Figura 6.11. Quitando MNF y VCA, que vimos no obtenían resultados de calidad adecuados y por tanto no son significativos aquí, vemos que PCA, ICA y SVD obtienen resultados similares, mientras que VQPCA ofrece menor compresión.

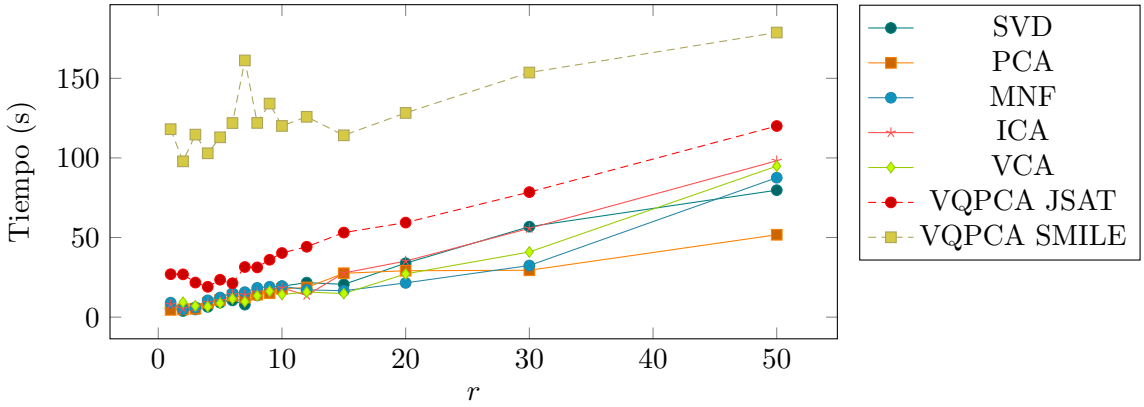


Figura 6.10: Tiempo de compresión respecto a r para los diferentes algoritmos implementados. (Compresión de REN).

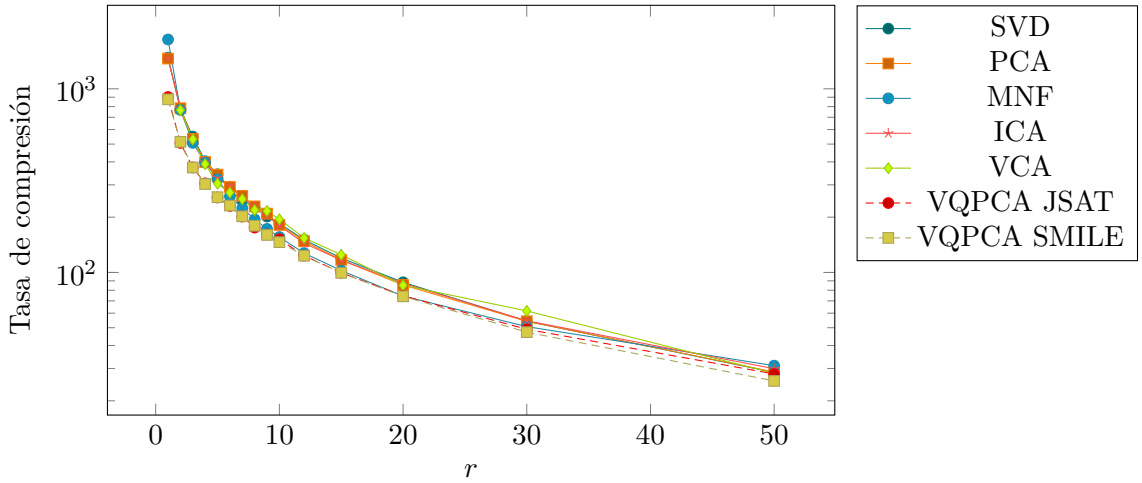


Figura 6.11: Tasa de compresión para los diferentes algoritmos según aumenta r. Nótese la escala logarítmica. (Compresión de REN).

6.5.1.1. ¿Con cuál nos quedamos?

Como hemos visto hasta ahora, esta es una pregunta difícil de responder, porque hay demasiados parámetros que entran en juego. ¿Es mejor lo más rápido? ¿Lo que ofrece mejor calidad? ¿Mayor tasa de compresión?.

Aunque no podemos definir una métrica numérica para ver cuál es el mejor de manera absoluta, sí podemos afirmar con rotundidad quién es mejor bajo ciertas restricciones. En concreto, si comparamos tasa de compresión con calidad, podemos decir que, fijada la tasa, es mejor aquel algoritmo que consiga una mejor calidad. Y de la misma manera, fijada una calidad, será mejor el algoritmo que la consiga con mayor tasa de compresión.

Así, vamos a comparar los algoritmos que aún no hemos descartado bajo estas condiciones en la Figura 6.12. Para tasas de compresión pequeñas, VQPCA es sin duda la opción ideal. Una vez superamos una tasa de compresión de ≈ 150 veces, PCA pasa a ser la opción nº1, porque la cabecera de VQPCA se hace demasiado pesada.

La rivalidad entre PCA y VQPCA desaparece al observar que, en realidad, PCA es igual a VQPCA con un único cluster. Nos interesa ver cuál es el número ideal de clusters para realizar la compresión. Mirando la Figura 6.13 vemos como para tasas de compresión pequeñas, lo ideal es utilizar un gran número de clusters, con una mejora notable hasta llegar a 4, estancándose después. Para compresión agresiva será mejor justo lo contrario: utilizar pocos clusters.

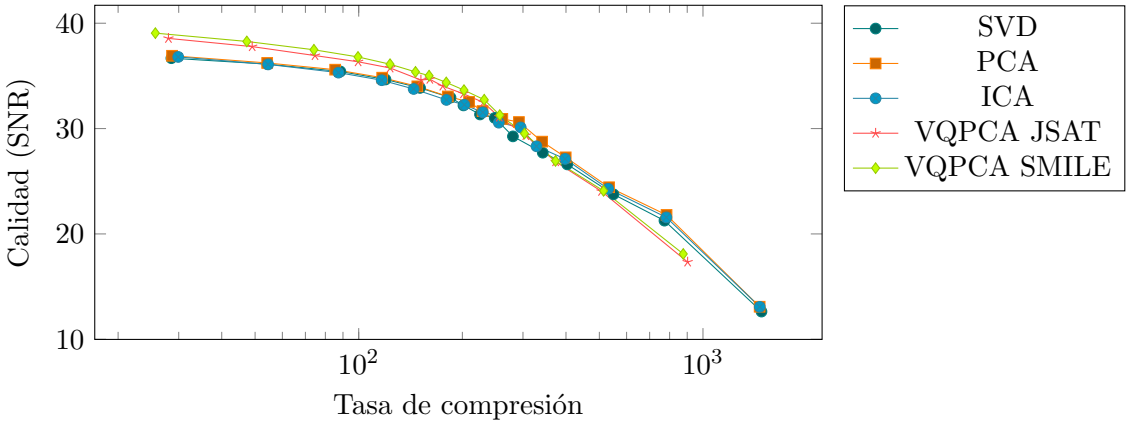


Figura 6.12: Comparativa de la compresión a cierta calidad para los diferentes algoritmos (Imagen REN).

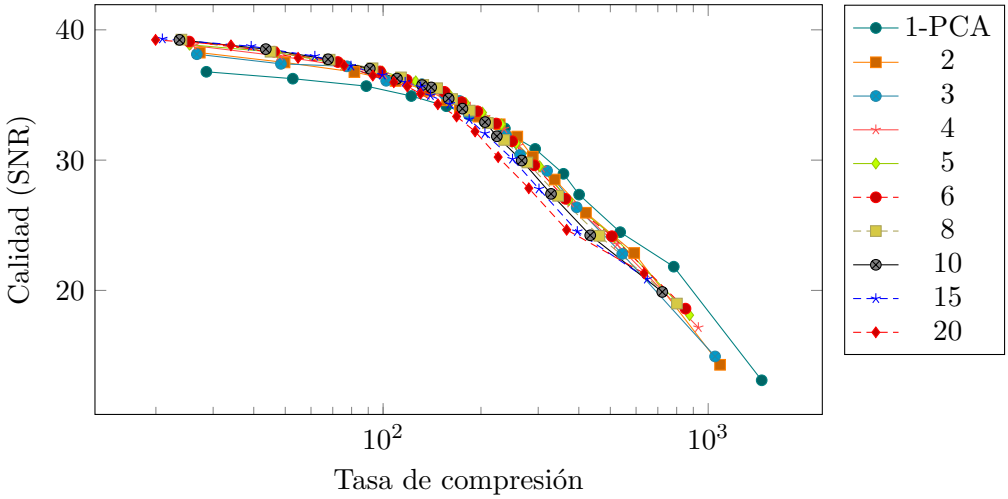


Figura 6.13: Rendimiento de VQPCA según el número de clusters elegido

Viendo esto, realizaremos el afinamiento de los demás parámetros utilizando un único cluster (caso en que VQPCA = PCA) a fin de acelerar los cálculos, buscando optimizar aún más las compresiones agresivas.

6.5.2. Aumentando la velocidad

Lo primero que queremos justificar es el uso de `-t 0.01`. Recordemos que esto indica que únicamente se utiliza el 1 % de las muestras de entrada para realizar el entrenamiento. A priori puede pensarse que esto puede inducir un gran error en la compresión, pues la reducción dimensional está ignorando mucha información.

Pero nada más lejos de la realidad. Las imágenes tienen una redundancia muy alta, y presentan muchos píxeles similares. Al escoger unos cuantos de manera aleatoria, vamos a tener un conjunto de entrenamiento muy similar (en cuanto a porcentajes de píxeles de ciertos valores), y por tanto los resultados de compresión serán similares (ver Tabla 6.3). De hecho puede incluso darse el caso de que la compresión da mejores resultados al reducir el conjunto de entrenamiento. Vemos un ejemplo en la Tabla 6.3, donde el ratio de compresión y SNR fluctúan sobre el mismo valor pese a cambiar t.

¿Por qué $t = 0{,}01$? El tiempo de ejecución no disminuye mucho más haciendo t menor, y así mantenemos cierta estabilidad del algoritmo, que podría perderse con imágenes de pequeño

t	0.001	0.005	0.01	0.03	0.05	0.1	0.3	0.5	0.75	1
Ratio	353.6	350.1	348.9	343.8	349.2	350.8	345.3	344.4	344.7	344.6
SNR	30.67	30.78	30.84	30.88	30.85	30.86	30.88	30.88	30.89	30.89
Tiempo (s)	5.402	5.673	5.918	6.353	6.891	8.338	14.22	20.08	27.62	37.63

Tabla 6.3: Resultados de la compresión de BEL utilizando diferentes valores de t. El tiempo es el único valor que cambia significativamente.

b		2	3	4	5	6	8	10	12	16
SNR	BEL	6.333	9.560	12.33	15.59	19.80	28.03	30.84	31.00	31.01
	SUW	-0.29	4.304	10.35	14.15	18.08	25.93	30.03	30.39	30.41
	CUP	-3.87	-0.77	6.881	11.20	13.39	19.82	23.19	23.45	23.46
Ratio	BEL	11458	11087	10191	7168	3472	796.0	348.8	225.1	135.2
	SUW	21371	20109	18642	13077	6268	1361	486.8	276.9	150.8
	CUP	3363	3314	3200	2975	2237	462.8	185.8	118.7	71.05
Time	BEL	03.26	02.53	02.62	02.78	02.80	03.82	05.52	07.16	10.19
	SUW	05.75	04.72	05.38	05.54	05.67	07.54	09.58	12.27	18.37
	CUP	01.43	01.14	00.91	00.96	01.17	01.62	02.84	03.87	05.83

Tabla 6.4: Resultados de la compresión de varias imágenes variando b

tamaño si hiciéramos t muy pequeña.

6.5.3. Profundidad de bits y calidad

Ahora queremos ver el efecto de b en la compresión. Partiendo de los parámetros por defecto, vamos a variar este parámetro con diferentes imágenes. Los resultados se muestran en la Tabla 6.4.

Saltan a la vista varias cosas. En primer lugar, ciertas imágenes tienen ruido superior a la señal para valores de b bajos ($SNR < 0$), por lo que querremos evitar cualquier cosa con $b < 4$ (además la calidad tampoco será buena). Por otro lado, vemos que a partir de $b = 8$ la calidad es aceptable, y a partir de $b = 10$ se estanca.

Valores $b > 10$ no aportan nada de cara a la calidad (ver Figura 6.14), y sin embargo afectan muy negativamente al ratio y al tiempo de compresión, que se dobla de $b = 10$ a $b = 16$.

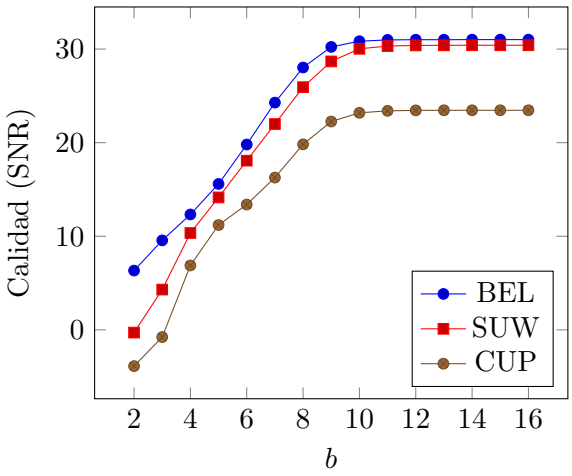


Figura 6.14: Calidad de imagen según aumenta b

Por tanto, si queremos la máxima calidad podemos elegir $b = 10$, bajando hasta $b = 8$ o incluso $b = 6$ en algunas imágenes consiguiendo calidades aceptables, y tasas de compresión extremadamente altas. (Por ejemplo, BEL con $b = 6$ tiene $SNR \approx 20$ (calidad aceptable) con un resultado 3472 veces menor que el original.

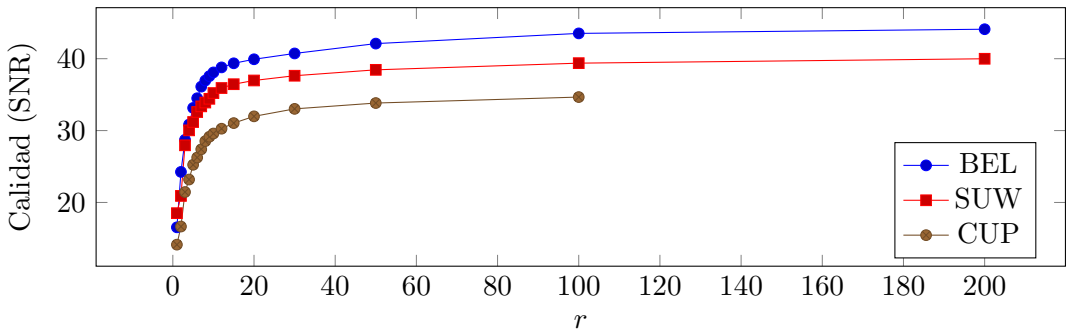


Figura 6.15: Calidad de imagen según aumenta r

r		1	2	3	4	5	8	15	30	100	200
SNR	BEL	16.54	24.25	28.72	30.84	33.15	36.96	39.36	40.73	43.52	44.10
	SUW	18.51	20.91	27.95	30.03	31.19	33.90	36.45	37.63	39.38	39.99
	CUP	14.13	16.65	21.45	23.19	25.24	28.53	31.04	33.02	34.66	n/a
Ratio	BEL	1157	684.8	464.8	348.8	272.7	162.4	79.60	35.97	9.733	5.100
	SUW	1616	858.9	666.3	486.8	336.8	199.4	84.85	39.44	10.89	5.542
	CUP	737.3	373.1	255.4	185.8	130.2	83.25	41.06	18.82	5.032	n7a
Time	BEL	04.14	03.98	04.61	05.80	07.12	11.64	17.40	36.77	2'08	4'09
	SUW	06.38	07.35	13.46	09.82	12.10	16.90	33.47	1'12	4'04	7'32
	CUP	02.71	02.02	02.45	02.99	03.64	05.22	10.06	20.98	23.30	n/a

Tabla 6.5: Resultados de la compresión de varias imágenes variando r

6.5.4. El impacto de la reducción dimensional

Ahora queremos ver hasta qué punto podemos aplicar reducción dimensional para no perder calidad. En la Figura 6.15 vemos cómo la calidad aumenta mucho con los primeros aumentos de r. En $r = 4$ ya se consiguen calidades aceptables para todas las imágenes, y a partir de $r = 15$ se observa ya poca mejora.

Pero aumentar r no es gratis. Si miramos la Tabla 6.5 observamos como el tiempo de compresión para los mayores r sube a varios minutos. El punto ideal está en el intervalo $4 \leq r \leq 15$, en el cual la compresión es bastante buena ($400 \rightarrow 50$ veces menos del tamaño original), el tiempo de compresión no es excesivo ($5 \rightarrow 40$ segundos), y la calidad es siempre lo suficientemente buena ($> 24dB$ SNR).

Se ha decidido utilizar $r = 4$ para las pruebas por comodidad de tiempo a la hora de generar gran cantidad de resultados. A la hora de comprimir una imagen para guardarla, lo ideal probablemente sea acercarse más a $r = 15$, manteniendo más características.

6.5.5. Dimensiones y bits

Hemos visto los efectos de cambiar b y r por separado, pero conviene ver qué comportamiento existe al mover ambos a la vez, para detectar posibles combinaciones que produzcan malos resultados.

En las Tablas 6.6 y 6.7 se muestran los resultados de distorsión y compresión para la imagen BEL en numerosas combinaciones de valores.

En primer lugar vemos que la calidad aumenta tanto con b como con r, por tanto será conveniente aumentar ambas variables de valor para aumentar la calidad. En general lo mejor es aumentar ambas en la misma cantidad partiendo de $b = 6, r = 2$.

$b\backslash r$	1	2	3	4	6	8	10	12	14	16
2	8,07	6,39	6,37	6,33	6,31	6,21	6,21	6,21	6,21	6,21
3	9,69	9,59	9,58	9,56	9,56	9,53	9,52	9,52	9,52	9,52
4	11,59	12,35	12,32	12,33	12,34	12,34	12,34	12,34	12,34	12,34
5	13,61	15,57	15,53	15,59	15,61	15,63	15,63	15,63	15,63	15,63
6	15,46	19,14	19,6	19,8	19,93	19,99	20	20,01	20,01	20,01
7	16,17	22,03	23,73	24,28	24,75	24,89	24,93	24,95	24,96	24,97
8	16,46	23,65	26,92	28,03	29,62	30,25	30,46	30,56	30,61	30,65
9	16,53	24,13	28,28	30,23	33,18	34,67	35,33	35,67	35,85	36,01
10	16,54	24,25	28,72	30,84	34,5	36,96	38,09	38,79	39,19	39,52
11	16,54	24,28	28,82	30,98	34,92	37,65	39,06	39,86	40,31	40,79
12	16,54	24,28	28,84	31	35,01	37,84	39,29	40,07	40,64	41,15
13	16,55	24,28	28,84	31	35,04	37,88	39,34	40,14	40,71	41,23
14	16,55	24,28	28,84	31,01	35,04	37,89	39,35	40,16	40,73	41,25
15	16,55	24,28	28,84	31,01	35,04	37,89	39,35	40,16	40,74	41,26
16	16,55	24,28	28,84	31,01	35,04	37,89	39,35	40,16	40,74	41,26

Tabla 6.6: Resultados de calidad (SNR) tras comprimir la imagen BEL con diferente número de bits y de dimensiones. La zona ideal es la verde claro. La parte azul tiene poca calidad, y la amarilla ocupará demasiado y tardará demasiado tiempo en generarse.

b\r	1	2	3	4	6	8	10	12	14	16
2	19383	15709	13251	11458	9018,7	7435,4	6325	5503,1	4870,34	4368
3	18104	15169	12848	11087	8746	7173,8	6124,9	5336,8	4738,79	4258,7
4	17803	14120	12033	10191	8027,8	6627,9	5576,4	4810	4252,75	3816
5	12899	10040	8650,2	7168,2	5284,8	4409	3457,3	2924,1	2583,49	2245,9
6	6601,7	5146,8	4220,1	3472,2	1960,4	1655,1	1247,8	964,17	814,14	662,3
7	3546,5	2610,4	1969,7	1562,5	835,95	676,57	498,6	380,78	315,71	259,06
8	2170,1	1467,9	1041	796,09	450,04	349,12	261,21	204,46	170,31	142,74
9	1510,9	946,25	650,47	489,97	291,81	222,08	169,56	135,45	113,76	96,69
10	1157,8	684,8	464,84	348,88	214,78	162,47	125,74	101,68	85,88	73,59
11	944,24	537,75	363,3	272,52	171,07	129,14	100,75	82,05	69,53	59,86
12	800,76	445,16	300,07	225,11	142,99	107,84	84,54	69,14	58,72	50,7
13	697,18	381,35	256,66	192,56	123,24	92,89	73,05	59,91	50,95	44,08
14	618,04	334,25	224,7	168,59	108,48	81,71	64,4	52,92	45,05	39,03
15	555,41	297,79	200	150,06	96,95	72,99	57,62	47,43	40,4	35,04
16	504,41	268,61	180,27	135,26	87,66	65,98	52,15	42,97	36,63	31,79

Tabla 6.7: Resultados de tasa de compresión tras comprimir la imagen BEL con diferente número de bits y de dimensiones. La curva amarilla es la zona ideal donde el tamaño no demasiado grande para no ser práctico (rojo) ni demasiado pequeño como para no tener calidad (verde).

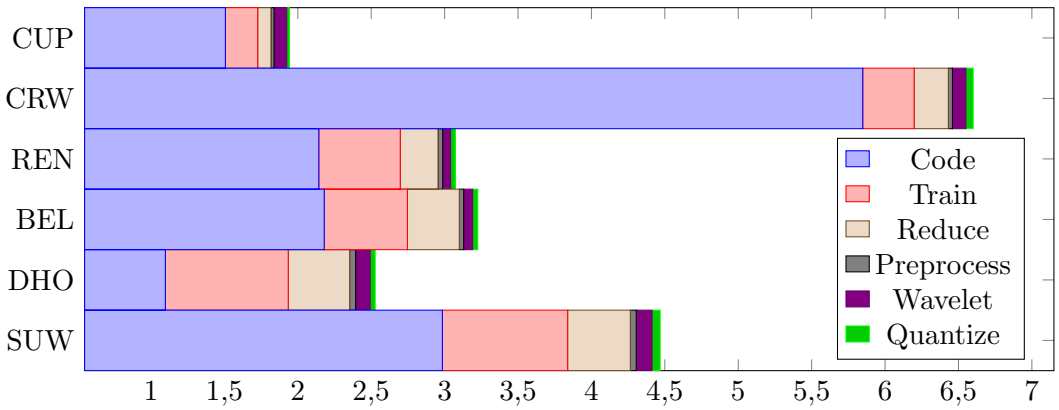


Figura 6.16: Profiling por tiempo de la compresión de las imágenes de muestra.

En cuanto al ratio de compresión, observamos que para calidades aceptables, los ratios rondan las 200 veces de compresión, conseguidos aproximadamente en la línea que une $b = 6, r = 16$ con $b = 16, r = 2$. Como en la primera tabla vimos que la calidad ideal se conseguía aproximadamente en la diagonal, los parámetros adecuados serán los que ronden $b = 9, r = 6$, según necesidades. De nuevo, aquí elegimos $b = 10, r = 4$ por la relación calidad/velocidad de procesado.

6.5.6. ¿A qué se dedica el tiempo?

Uno de los objetivos de este algoritmo es conseguir una ejecución rápida. A lo largo de las secciones anteriores hemos visto como PCA (o VQPCA con 1 cluster) era el algoritmo ideal para la reducción. Rápido y con la mejor calidad en la mayor parte de los casos. Nuestro interés es ahora acelerarlo lo más posible. Ya hemos hecho gran parte con la elección de $t = 0{,}01$. Veamos cómo se distribuye el tiempo de ejecución para la compresión de todas las imágenes de muestra con los parámetros por defecto. La Figura 6.16 muestra los resultados.

Es claro ver que las partes más importantes de la compresión son la codificación en bloques, el entrenamiento del algoritmo de reducción de dimensionalidad, y la reducción en sí. La codificación tarda, con diferencia, el mayor tiempo de las tres, ocupando habitualmente unos dos tercios del tiempo total, y llegando hasta el 80 % en las imágenes con menos bandas y que por tanto requieren menos tiempo de reducción.

Esta parte es, por suerte, una de las más paralelizables de todo el algoritmo: la codificación de cada bloque se puede hacer de manera independiente. Además, la codificación se realiza íntegramente con aritmética de números enteros y operaciones a nivel de bit.

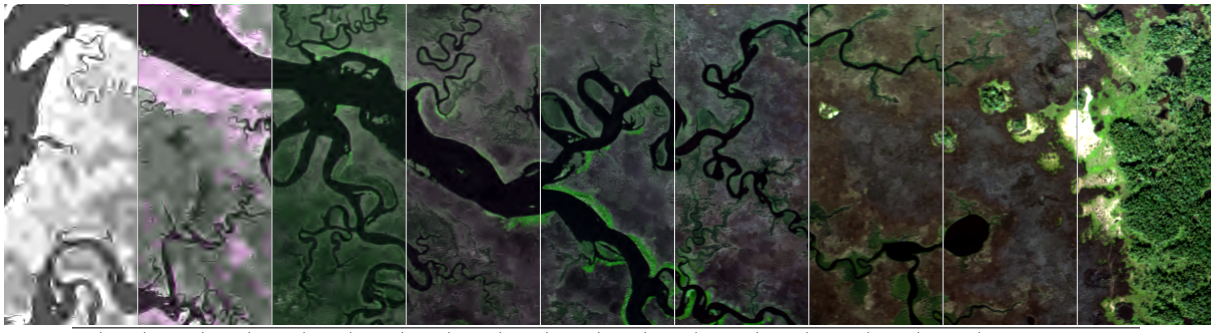
Una FPGA es la plataforma ideal donde acelerar este proceso: De lo más rápido para operaciones lógicas y enteras sencillas, pues las ejecutamos directamente sobre silicio en lugar de mediante instrucciones, y con la capacidad de paralelizar masivamente los algoritmos que ejecuta, ya que podemos replicar un módulo tantas veces como quepa en la placa.

Así pues, en el Capítulo 7 diseñaremos un módulo de compresión por bloques para FPGA, para ver hasta dónde podemos acelerar el algoritmo.

6.5.7. Calidad de las imágenes comprimidas

Hasta ahora hemos visto todos los resultados de manera numérica y teórica pero, ¿qué representan en realidad estos números?. En la Figura 6.17 observamos la evolución de la imagen desde la compresión más agresiva hasta una de las que más calidad mantienen, variando b y r.

Las tres primeras muestras mantienen cierta estructura espacial, pero pierden todo dato de



(b, r)	$(5, 1)$	$(6, 2)$	$(7, 3)$	$(8, 4)$	$(9, 5)$	$(10, 6)$	$(11, 7)$	$(12, 8)$	Original
SNR	13.55	16.78	21.81	25.93	29.76	32.58	34.01	35.07	∞
Ratio	22237	8529	3653	1361	490.7	261.3	174.2	122.8	1
Time	1.322	1.660	1.992	2.806	4.399	5.384	8.079	13.73	0

Figura 6.17: Evolución de la calidad con la compresión de la imagen.

`-s`	0 0	3 1	2 1	1 1	1 1 3 2	1 1 2 2	1 1 2 2 3 3	1 1 3 3
Calidad (SNR)	30.84	30.84	30.70	30.56	30.54	30.05	30.01	30.47
Ratio de compresión	348.8	376.6	409.1	445.8	491.4	545.3	598.6	534.3

Tabla 6.8: Vemos que según reduzcamos los bits en diferentes bandas, la calidad se mantiene más o menos similar a la original. Resultados para la imagen BEL.

la estructura espectral al reducir demasiado las dimensiones. Las tres siguientes ya resuelven correctamente los detalles espaciales, y al nivel espectral se acercan bastante a la realidad, pero se nota que no son totalmente correctas. Las tres últimas (entre las que se encuentra la original) son prácticamente indistinguibles al ojo, y se necesitaría un software específico para diferenciarlas adecuadamente.

En resumen, los resultados de compresión son:

- Ratio $>$ 2000: Pérdida de información espectral, se resuelve más información espacial cuanto menos se comprime.
- Ratio 200 $\rightarrow$ 2000: Información espacial resuelta, la información espectral crece conforme menos comprimimos.
- Ratio 100 $\rightarrow$ 200: Imagen indistinguible a simple vista.

Por supuesto existen muchas más opciones que cambian estos comportamientos, pero por regla general podremos comprimir una imagen entre 100 y 1000 veces sin dificultades.

6.6. Profundidad de bits variable

Los algoritmos de reducción dimensional suelen concentrar la información más importante en las primeras dimensiones, dejando las últimas como un mero refinamiento de las primeras. Hasta ahora siempre hemos aplicado la misma profundidad de bits en todas las dimensiones reducidas, pero con la opción `-s` podemos configurar el algoritmo para aplicar diferentes profundidades en cada dimensión.

En la Tabla 6.8 vemos los resultados de ir quitando progresivamente más bits a las dimensiones menos importantes. Pero queda claro que los resultados, si bien excelentes en algunas configuraciones, son algo erráticos.

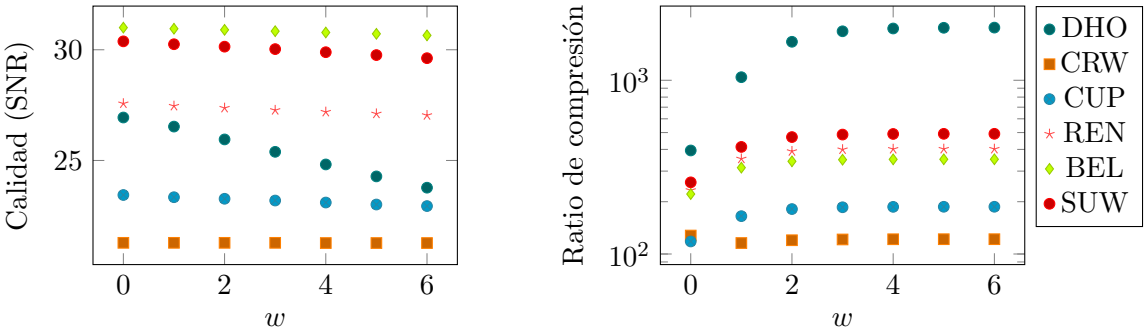


Figura 6.18: Se observa cómo varían la calidad y el ratio de compresión según las pasadas de ondícula.

Porcentaje en crudo	0	0.00001	0.0001	0.001	0.01
Calidad	30.844	30.85	30.88	30.92	30.94
Ratio de compresión	348.8	345.6	331.6	298.8	245.2

Tabla 6.9: Se observa cómo varían la calidad y el ratio de compresión según las pasadas de ondícula para la imagen BEL.

Por ejemplo, al quitar dos bits de la tercera dimensión[3] (cuando se ejecuta **`-s 1 1 2 2`**) la calidad baja casi medio dB. Sin embargo, quitando un bit más a la tercera dimensión con **`-s 1 1 3 3`** no tenemos ese problema.

Es decir, que este ajuste, si bien puede conseguir muy buenos resultados, debe hacerse a mano y con cuidado, conociendo las particularidades de cada imagen, y casi por prueba y error. No podemos dar una configuración global para todas las imágenes.

6.7. Otros parámetros menos importantes

6.7.1. Modificando la ondícula

Desde el comienzo estamos utilizando la opción **`-w 3`**, que quiere decir que aplicamos tres pasadas de ondícula en el algoritmo. Ese número puede cambiar, y podemos hacer tantas pasadas como queramos. En la Figura 6.18 observamos los efectos sobre la compresión y calidad de modificar este parámetro.

Por regla general, la tasa de compresión mejora a costa de una pequeña pérdida de calidad. Por tanto nos saldrá a cuenta aplicar la transformada. Hasta $w = 3$ suele mejorar bastante, y a partir de ahí estancarse. De ahí la elección.

6.7.2. Codificación en crudo de anomalías

Mediante la opción **`--hardcode_outliers`** se puede codificar una parte de las muestras de los extremos en crudo antes de la cuantización, a fin de reducir el intervalo de cuantización y mejorar por tanto la resolución a la hora de codificar. En el conjunto de imágenes de prueba esto no ha tenido ningún efecto notable. La calidad mejora marginalmente a un coste de compresión que no es aceptable. A modo de ejemplo se muestran unos resultados en la Tabla 6.9.

[3]Los índices están basados en cero, por eso que 2 sea la tercera dimensión

6.8. Comparativa con algoritmos existentes

Existen muchas aproximaciones diferentes para la compresión de imágenes hiperespectrales, si bien las mejores técnicas [54] a día de hoy son las que incluyen algoritmos que aprovechen la correlación espectral de las imágenes. Aunque los algoritmos de análisis y clasificación no obtienen los mismos resultados [73] con imágenes comprimidas y sin comprimir, en ocasiones el suavizado derivado de la compresión es beneficioso para las clasificaciones. Los algoritmos de compresión son por tanto bienvenidos en los estudios.

La comparativa más directa la podemos establecer con el trabajo de [74], donde utilizan la misma medida de SNR que en este trabajo, obteniendo resultados de unos 10-20dB para tasas de compresión de 160, mientras que aquí obtenemos, para la misma compresión, resultados en el rango de los 20-30dB. Los algoritmos que estudian incluyen SPIHT y SPECK, que tras una transformación de ondícula tridimensional codifican la imagen mediante árboles también tridimensionales, que progresivamente refinan "cubos" de la imagen hiperespectral. Además estudian la versión de JPEG2000 especificada en el estándar con transformación total en el espectro. Vista la diferencia con nuestra versión, parece buena decisión no haber utilizado la reducción dimensional incluida en el estándar. En [53] proponen una mejora más sobre SPECK (el mejor de los tres algoritmos) que aun así presenta peores resultados de distorsión que PCA+JPEG2000 (13-22dB).

Por otro lado tenemos la implementación en cuya idea nos basamos [20], que combina PCA + JPEG2000. Por desgracia utilizan otra fórmula para el cálculo del SNR, por lo que no podemos establecer comparativas directas. Sin embargo nuestros resultados (Figura 6.13) mostraron que nuestra aproximación con VQPCA es algo superior en tasas de compresión bajas, y algo inferior en tasas de compresión altas, por lo que dependiendo de la situación será mejor uno u otro algoritmo. Otro trabajo de los mismos autores [75] ratifica la idea de realizar submuestreo para el entrenamiento de PCA, con resultados muy similares entre las dos versiones. También propone un método aproximado de calcular PCA que acelera aún más los cálculos. Sería interesante en un futuro ver si puede aplicarse aquí la misma idea.

Finalmente, los resultados de [52] muestran un rendimiento superior que nuestra implementación para calidades de imagen alta, donde consiguen una compresión mejor a partir de los 40dB. La clave de su éxito radica en detectar la importancia de las diversas bandas, y realizar una compresión progresiva utilizando JPEG2000, que comienza codificando la información más importante según distorsión. Mezclar estas técnicas en un futuro con la reducción dimensional podría mejorar aún más el rendimiento del algoritmo.

Capítulo 7

EBCoder sobre FPGA

La parte más costosa del algoritmo de compresión es la codificación por bloques. En cada bloque la codificación debe seguir un orden estricto, y debemos recorrer exhaustivamente el bloque de principio a fin para asegurar que queda completa y correctamente codificado.

La totalidad de las operaciones de codificación tratan con números enteros, con tipos enumerados, o con valores *booleanos*. Un procesador de propósito general no aprovecha la simplicidad de estos datos, teniendo demasiada maquinaria inactiva cuando trata con ellos.

Sin embargo, una FPGA se puede amoldar con precisión a las necesidades del algoritmo. Todas las operaciones que en un procesador utilizan 32 o 64 bits pueden ahora hacerse con solo 3 si es necesario, con el incremento de velocidad que ello supone. Y no solo eso, sino que la FPGA es capaz de ejecutar todos los cálculos simultáneamente, y comprimir a razón de una muestra por ciclo al eliminar las latencias introducidas por un código con ejecuciones condicionales.

Así pues, pasamos a diseñar una arquitectura del EBCoder para FPGA, que observamos en la Figura 7.1. En resumidas cuentas, creamos memorias para cada variable de estado de las muestras codificadas, un búfer para los datos de entrada, y la lógica necesaria que interpreta y comprime los datos. Todo esto se recoge en un repositorio de código VHDL llamado VYPEC [76] (Vhdl jYPEC).

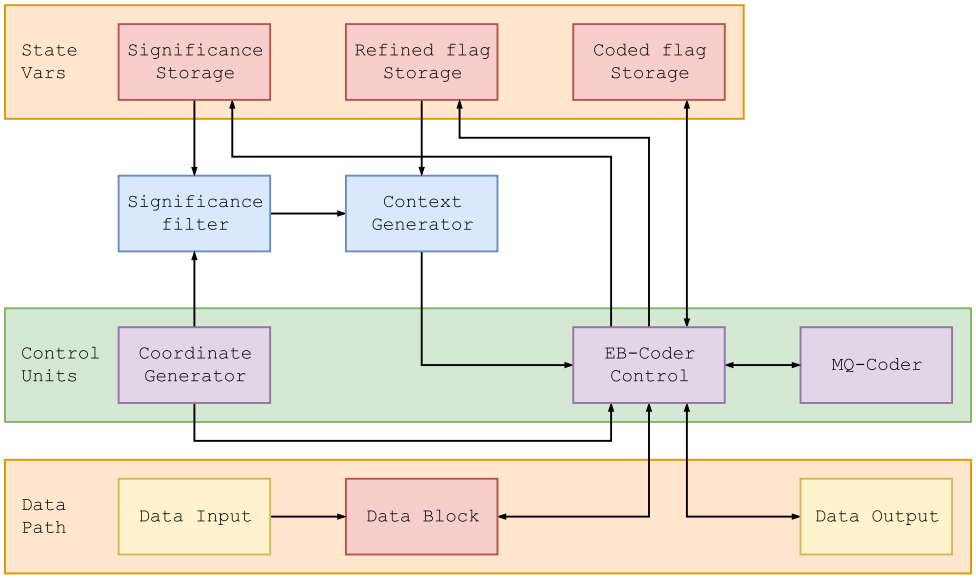


Figura 7.1: Esquema general de los diferentes módulos del EBCoder. Se distinguen las memorias, los módulos combinacionales, los módulos mixtos y las entradas y salidas

En las siguientes secciones veremos más en detalle cómo funciona cada módulo y las depen-

dencias entre ellos.

7.1. Bloque de datos

En primer lugar tenemos el almacenamiento de datos. Recordamos del Capítulo 4-Sección 4.2.4 que damos varias pasadas por los diferentes planos de bits del bloque hasta que está totalmente codificado. Los datos no nos van a venir bit a bit, sino completos, por lo que tenemos que guardar las muestras enteras para ir recuperando en cada momento las necesarias. Posteriormente un filtro decidirá qué bit de la muestra completa es el que necesita.

¿Y cuáles son necesarias? La muestra actual siempre es necesaria, y en el peor de los casos (la codificación en carrera), también lo serán las tres siguientes. Así pues, necesitaríamos una memoria con lectura cuádruple.

Sin embargo, podemos simplificar mucho el asunto dándonos cuenta de que las pasadas son siempre en el mismo orden, por lo que podríamos utilizar una cola FIFO (ver Figura 7.2) que en cada momento nos de la muestra adecuada. Como con la cola no podemos obtener las tres muestras siguientes, lo que haremos será mantener las tres anteriores en un registro de desplazamiento. Adelantando la cola tres posiciones al arranque del algoritmo, tendremos las muestras requeridas. Hay que notar que esto solo es posible porque el bloque es inmutable, y sus valores no cambian.

El uso de una cola también es beneficioso a la hora de leer los datos, que llegan en serie desde el exterior.

Así, en el bloque de datos mantendremos dos índices: Uno que indica cuántas muestras tenemos disponibles, y otro que indica hasta dónde hemos comprimido. Si la velocidad de entrada es pequeña, comprimiremos las muestras más rápido de lo que llegan. Para evitar comprimir información inválida, una señal se levantará indicando que no hay muestras disponibles si el índice de lectura se iguala al de escritura, y parará el EBCoder hasta que haya más disponibles. Una vez se haya escrito el bloque entero, el índice de escritura se desactiva, y se continúa leyendo pasada tras pasada.

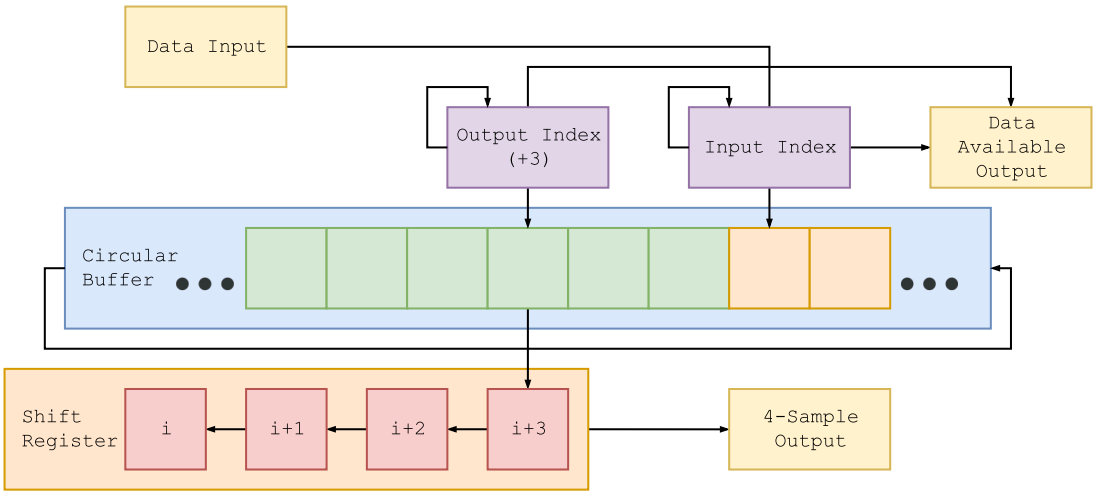


Figura 7.2: Esquema general del bloque de datos. El índice de entrada se desactiva al llegar al final. El índice de salida da la vuelta tantas veces como pasadas sean necesarias. La lectura va tres ciclos adelantada para poder extraer la muestra actual y tres siguientes.

Por simplicidad posterior, las muestras se guardan ya en zig-zag en el búfer circular.

7.2. Generación de coordenadas

Numerosas funciones del codificador dependen de la posición del bloque que estemos codificando: Los entornos se truncan en los bordes, la codificación de carrera se hace únicamente al comienzo de una subcolumna, etc.

El generador de coordenadas (Figura 7.3) se encarga de indicar en cada momento la posición en el bloque, así como la pasada que estamos dando al plano de bits, y qué plano de bits es el que estamos codificando. Además genera un indicador al llegar al final para detener la codificación una vez se ha terminado.

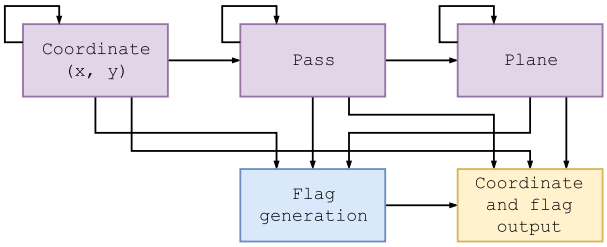


Figura 7.3: El contador incrementa por orden la posición, pasada y por último el índice de plano de bits. Además los utiliza para generar los marcadores de terminación

7.3. Almacén de significancia

Dependiendo de la posición de la muestra que estamos codificando, puede ser necesario saber el estado de significancia de hasta 18 muestras diferentes para realizar la codificación adecuadamente. Para solucionar este problema se utiliza una técnica similar a la del bloque de datos.

Pueden ser necesarias muestras de la tira en zig-zag superior e inferior, pero guardar dos tiras enteras en registros de desplazamiento sería excesivo. En la Figura 7.4 vemos la solución, que consiste en tener una memoria principal y dos secundarias, interconectando registros de desplazamiento que contienen las muestras que actualmente están o van a estar en uso.

Un diagrama más detallado de qué muestras se mantienen en memoria puede verse en la Figura 7.5

7.4. Filtro de significancia

Hemos visto que el almacén de significancia genera una gran cantidad de datos que, observando la Figura 7.5, vemos que no siempre significan lo mismo. Por ejemplo, en diferentes

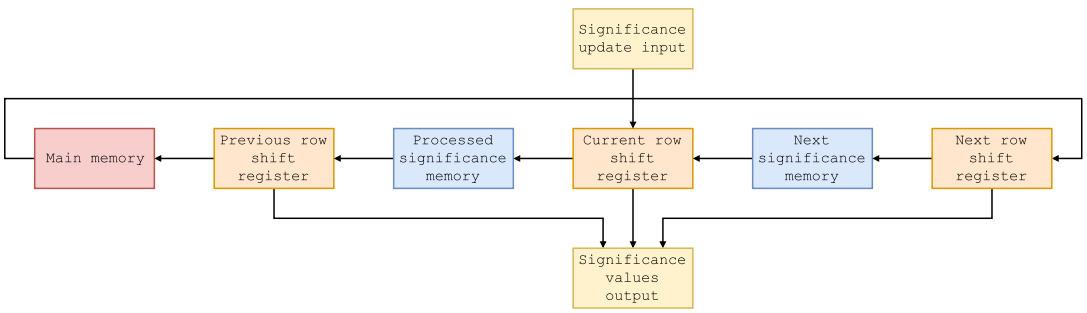


Figura 7.4: Las diferentes memorias e interconexiones del almacén de significancia

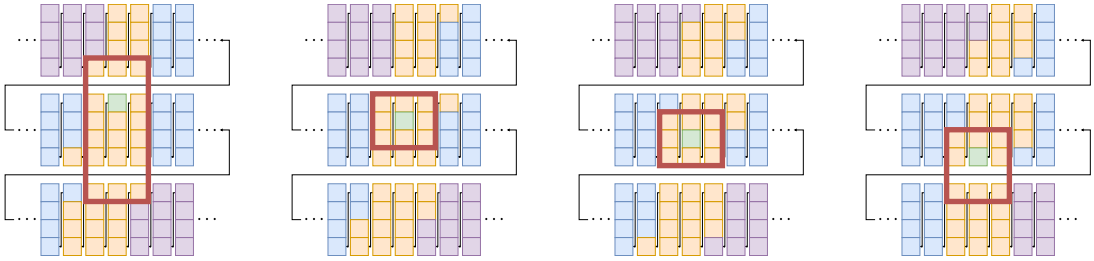


Figura 7.5: Estructura detallada de las memorias de significancia y sus conexiones. Vemos las cuatro posibles situaciones que se pueden dar, y qué muestras son necesarias en cada una.

situaciones el elemento inmediatamente anterior en el registro de desplazamiento central puede ser la muestra superior, o situarse a la izquierda y tres posiciones más abajo. Y no solo eso, cuando estamos en los bordes del bloque los valores que nos devuelva el almacén no serán los que esperamos, por ejemplo al estar en la primera fila y no disponer de fila anterior.

Para solucionar estos problemas se introduce el filtro de significancia, que según la coordenada en la que estemos va a decir qué salida del almacén corresponde a qué posición del vecindario. Además, en caso de estar en los bordes del bloque sustituirá los valores de significancia del almacén por el valor por defecto.

7.5. Almacén de refinamiento

El almacén de refinamiento guarda un bit por muestra que indica si ya ha sido refinado o no. Funciona como una cola FIFO, mucho más simple esta vez pues únicamente tiene una entrada y salida.

7.6. Generación de contexto

La generación de contexto parte del vecindario de significancia ya filtrado, y del bit de refinado, para generar el contexto actual. Existen diferentes formas de calcular el contexto, dependiendo de la fase de la compresión en la que estemos (Sección 4.2.4). El generador de contexto calcula y devuelve todas, delegando en el EBCoder la elección del contexto apropiado.

Por otro lado, el generador de contexto levanta una señal si detecta que todas las muestras de una subcolumna son insignificantes, para ayudar en la codificación de carrera.

7.7. Almacén de codificación

Es idéntico al de refinamiento, guardando un bit que indica, para cada bit de un plano, si ha sido o no codificado. En lugar de guardar para cada muestra tantos bits como tiene, lo que se hace es limpiar el almacén de codificación en la pasada de limpieza, y reutilizarlo en la siguiente pasada de significancia, a la que llega limpio.

7.8. Control del EBCoder

La unidad de control es la que orquesta el funcionamiento de todos los demás módulos, activando o desactivándolos según sea necesario. Está gobernada por la máquina de estados de

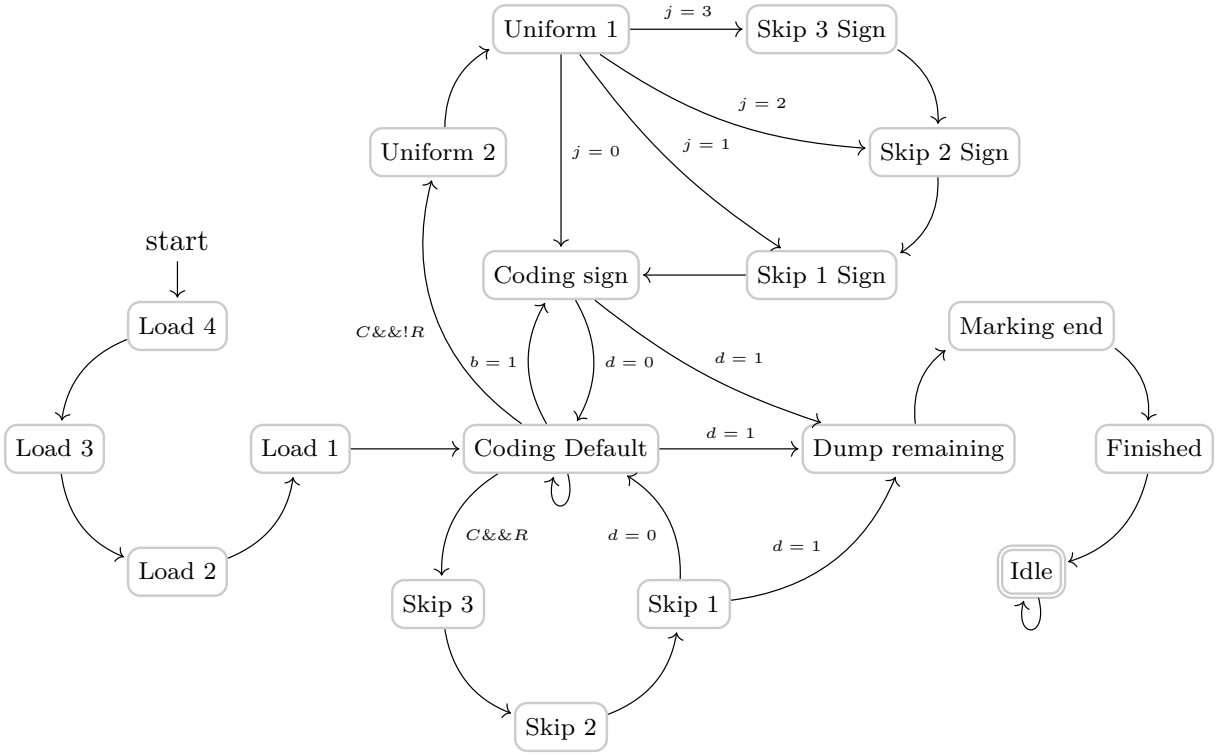


Figura 7.6: Máquina de estados de la unidad de control del EBCoder. Las transiciones son explicadas en la Sección 7.8

la Figura 7.6, donde se muestran todas las transiciones existentes. Los valores por defecto, que cada estado podrá sobreescribir, se muestran en el Algoritmo 17.

- LOAD_4 - LOAD_1: El estado inicial es LOAD_4. Estos cuatro estados se encargan de precargar el registro de desplazamiento del bloque de datos (Sección 7.1). Durante cuatro ciclos se activa memory_shift_enable, y así se consigue disponer de la primera muestra, así como las tres siguientes, cuando pasamos al estado de codificación.

  Estos cuatro estados desactivan la generación de coordenadas utilizando coordinate_disable, para que al terminar de cargar, la posición inicial sea la $(0, 0)$ en pasada de limpieza para el primer plano de bits.

- CODING_DEFAULT: Es el estado principal desde donde se decide qué hacer con cada bit, dependiendo de la pasada en la que estemos:

  - Significancia: En caso de necesitar codificar el bit actual lo enviamos junto al contexto al codificador aritmético en mqcoder_bit y mqcoder_context, activando mqcoder_enable. Si es un **1**, pasamos al estado CODING_SIGN, y en caso contrario seguimos en CODING_DEFAULT. Si no hace falta codificarlo, simplemente hacemos una petición a la memoria para que pase al siguiente valor mediante memory_shift_enable, y continuamos en el mismo estado.
  - Refinamiento: En caso de refinarse el bit, se actualiza el marcador next_refinement_flag y se activa el codificador aritmético. El siguiente estado siempre será CODING_DEFAULT tras actualizarse las memorias con memory_shift_enable.
  - Limpieza: El caso de limpieza es más complicado. En primer lugar, puede ocurrir que se requiera utilizar la codificación en carrera.

    En este caso puede pasar que nos la interrumpa un bit **1** o no. Esta condición es indicada al codificador aritmético con CONTEXT_RUN_LENGTH, y se pasa bien a

Algoritmo 17: Valores por defecto en EBCoder

```
1  next_state ← current_state ;                          /* mantenemos estado */
2  memory_shift_enable ← 0;  /* no activamos desplazamiento de memoria */
3  coordinate_disable ← 0; /* no desactivamos generación de coordenadas */
4  mqcoder_enable ← 0;          /* desactivamos codificador aritmético */
5  mqcoder_bit ← 0;
6  mqcoder_context ← CONTEXT_UNIFORM;
7  mqcoder_end_coding ← 0;
8  if estamos en la pasada CLEANUP then
9      is_coded_flag_next ← 0;                        /* reseteamos marcador */
10 else
11     is_coded_flag_next ← is_coded_flag_current;
12 end
13 if estamos en la pasada CLEANUP del primer plano then
14     next_significance ← INSIGNIFICANT ;      /* reseteamos significancia */
15     next_refinement_flag ← 1;                 /* reseteamos refinamiento */
16 else
17     next_significance ← current_significance;
18     next_refinement_flag ← current_refinement_flag;
19 end
20 output ← mqcoder_output ;          /* sale el codificador por defecto */
21 output_enable ← mqcoder_output_enable;
22 busy ← 1;                            /* ocupados hasta que terminemos */
```

saltar las siguientes muestras (que son cero) en los estados SKIP_3 - SKIP_1, o a codificar el error de carrera en los estados UNIFORM_2 y UNIFORM_1.

Por otro lado, si no hace falta la codificación de carrera, el bit será codificado como si fuera de significancia, pasando a codificar signo si fuera necesario.

En caso de acabarse la codificación (recordemos que la limpieza es la última pasada) se cambia al estado DUMPING para vaciar el codificador aritmético.

- CODING_SIGN: Aquí codificamos el signo correspondiente a una muestra con el contexto especial de signo. Enviamos al codificador aritmético el bit y contexto adecuados levantando mqcoder_enable, y actualizamos la significancia actual de la muestra en next_significance (positiva o negativa). Además marcamos el bit como codificado con is_coded_flag_next. En caso de estar en la última posición, saltamos a DUMPING. De lo contrario volvemos a CODING_DEFAULT.

- SKIP_3 - SKIP_1: En caso de hacer una codificación en carrera de una subcolumna donde todo son ceros, debemos saltar cuatro muestras hasta llegar a la siguiente columna. Una es consumida por CODING_DEFAULT, y las otras tres en estos estados. Tras consumirse se vuelve a CODING_DEFAULT y se prosigue con las muestras siguientes, a menos que se hayan acabado en cuyo caso se salta a DUMPING.

  Durante todos ellos se activa memory_shift_enable.

- UNIFORM_2 y UNIFORM_1: A estos estados se salta en caso de interrumpirse una codificación en carrera: Se creía que las cuatro muestras de una subcolumna iban a ser ceros, pero una de las cuatro rompió la racha. En los dos estados se codifica el índice que rompió la racha, mandando al codificador aritmético la codificación en binario del índice ($0 \rightarrow \mathtt{00}, 1 \rightarrow \mathtt{01}, \ldots$) en mqcoder_bit, y el contexto CONTEXT_UNIFORM en mqcoder_context. Por supuesto se levanta mqcoder_enable para activar el codificador.

A continuación se debe codificar el signo de la muestra en cuestión, y seguir tras la misma codificando de manera usual. Dependiendo de qué muestra j fue la que rompió la racha, puede hacer falta saltar unas pocas hasta llegar a ella ($j > 0$), para lo que se saltaría a uno de los estados SKIP_3_SIGN - SKIP_1_SIGN. En caso de no ser necesario, se salta a CODING_SIGN directamente ($j = 0$).

- SKIP_3_SIGN - SKIP_1_SIGN: Estos estados son análogos a los estados SKIP_3 - SKIP_1, solo que al terminar saltan a CODING_SIGN. Se utilizan para codificar el signo de la muestra que rompe la codificación de carrera. (Nótese que el bit de magnitud no se codifica pues siempre que se rompe la codificación de carrera lo hace un **1**, por eso se salta directamente al signo).

  Durante todos ellos se activa memory_shift_enable.

- DUMPING: Indicamos al codificador aritmético que hemos terminado de codificar el bloque levantando mqcoder_end_coding (recordemos que el codificador aritmético solo recibe parejas (bit, contexto), y por tanto no sabe por dónde se va codificando). El codificador aritmético vaciará así sus búfer en mqcoder_output para completar el flujo de bits comprimidos. Pasamos directamente al estado de MARKING_END.

- MARKING_END: En este estado escribimos los bits sobrantes que pedimos al codificador aritmético en **dumping** y que se encuentran en mqcoder_output y mqcoder_output_enable, y pasamos directamente a FINISHED.

- FINISHED: Ya hemos terminado la codificación aritmética, así que sacamos el código especial de terminación `0xfffe` y pasamos al estado IDLE.

- IDLE: El estado de espera. No se hace nada hasta que se reciba una señal de reset, en cuyo caso se volverá a LOAD_4.

7.9. Codificador aritmético MQ

Ya tenemos generados el flujo de bits y contextos que se van a utilizar para la codificación. El codificador MQ es el encargado de, partiendo de ellos, crear el flujo comprimido de bits.

El primer problema que nos planteamos es cómo hacer un diseño capaz de procesar rápido las entradas. Un vistazo al Algoritmo 15 evidencia esta dificultad: Tenemos un bucle con un número indefinido de iteraciones que no podemos desenrollar trivialmente.

La renormalización funciona duplicando el valor de un registro de 16 bits hasta que el bit más significativo es **1**. Así pues una cota superior son 15 ejecuciones del bucle. Pero desenrollar 15 vueltas es poco eficiente. En cada vuelta, un bit es emitido por la salida. ¡Pero nosotros trabajamos con bytes!. Si en lugar de duplicar el valor lo multiplicamos por hasta 256, conseguiremos acelerar en un factor de 8 la ejecución.

Así pues colocamos un búfer de 8 bits en el codificador, que cada vez que se llene será vaciado (hay que tener en cuenta que la codificación de un símbolo nunca suele añadir un número de bits múltiplo de 8, por lo que es imprescindible un búfer). Ahora tenemos que ver cuál es el número máximo de veces que puede llenarse en un ciclo:

El caso peor es que contenga ya 7 bits. Añadimos un nuevo bit y vaciamos el búfer. Pero surge un pequeño problema: si el byte emitido es `0xff`, debemos añadir un **0** para evitar que aparezca en el flujo la secuencia `0xfffx`. Dichas secuencias están reservadas por el estándar JPEG2000 para, entre otras cosas, indicar al decodificador la terminación del bloque. Por tanto solo puede aparecer al final del flujo, y no entre medias.

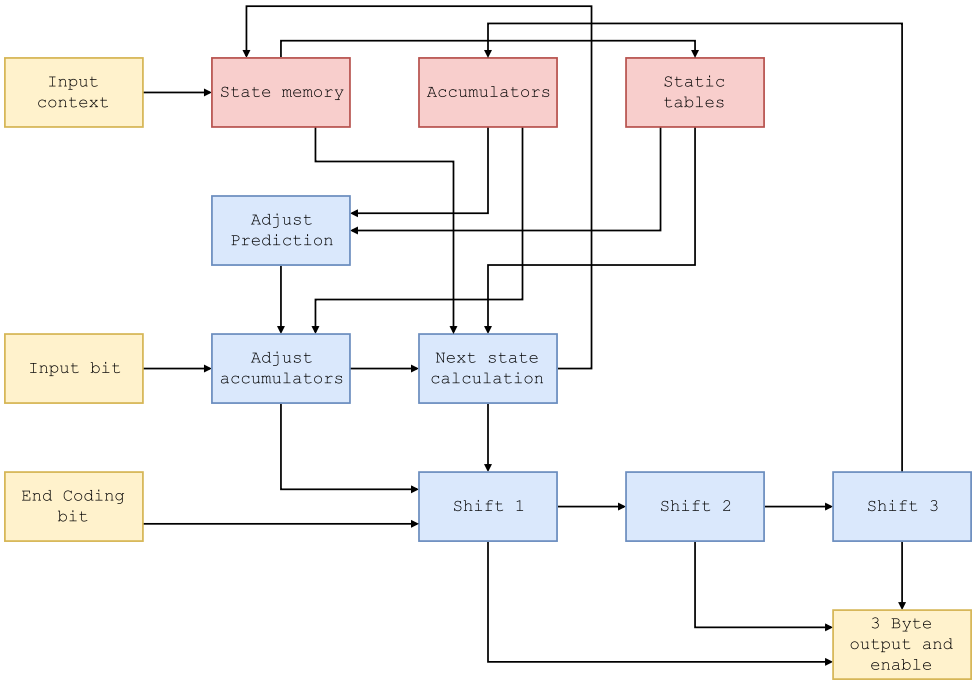


Figura 7.7: Esquema de los diferentes módulos del codificador aritmético. Pese a necesitar bucles de renormalización en software, se consigue que procese un símbolo cada ciclo de reloj en hardware.

En cualquier caso, añadimos el bit extra, y, en el caso peor, tendremos que poner otros 7 hasta llenar el búfer, y emitir un byte de nuevo, que de seguro no puede ser `0xff`. Así, nos quedan $15 - 7 - 1 = 7$ bits en el peor de los casos, que entrarán sin problema en el búfer, teniendo un hueco extra para el siguiente ciclo.

En total sumamos hasta tres desplazamientos y dos emisiones de byte.

Existe un caso especial, al terminar la codificación, donde es obligatorio terminar de vaciar el búfer para no perder esos bits temporales. En ese caso sí necesitaremos, tras el último desplazamiento, emitir un byte extra. Así pues colocaremos, para estos casos especiales, un tercer byte de salida que sólo podrá aparecer activo en la terminación de codificación. Para indicar los bytes válidos de la salida, se incluyen 3 bits extras que se ponen a **1** cuando es válido el byte asociado.

En resumen, un desenrollado que ingenuamente habría supuesto 15 fases distintas lo conseguimos hacer en 3, que pueden colocarse una tras otra combinacionalmente para alcanzar el procesado de una pareja (bit, contexto) por ciclo. Podemos ver la estructura del módulo en la Figura 7.7.

7.10. Empaquetando el módulo

Ahora mismo tenemos un módulo que cada ciclo es capaz de asimilar una muestra de entrada, hasta llenar el bloque de datos, y mientras tanto ir produciendo, también cada ciclo, hasta 3 bytes de información.

Para facilitar la integración del módulo con otros componentes es deseable que la entrada y la salida sean byte a byte. Para ello se ha empaquetado junto a otros módulos que ocultan el funcionamiento al exterior. El supermódulo resultante (Figura 7.8) muestra al exterior los siguientes puertos:

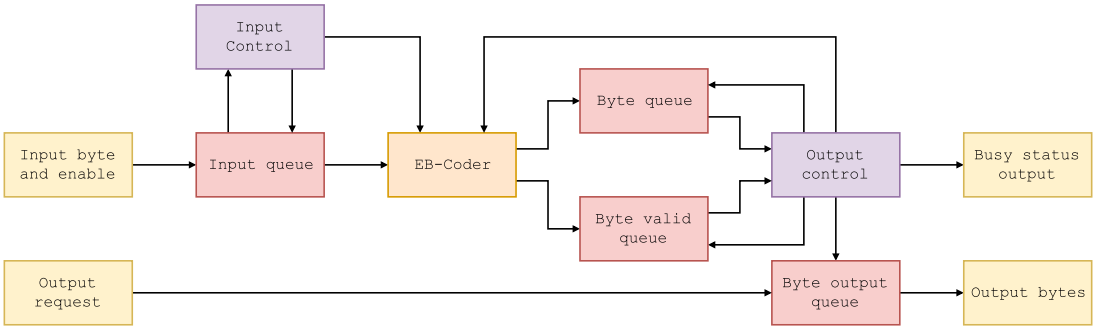


Figura 7.8: Esquema del supermódulo que contiene al EBCoder. Este supermódulo oculta el funcionamiento interno y ofrece una interfaz de entrada y otra de salida de 8 bits de anchura.

- Entradas de control clk y rst.
- Una entrada input de 8 bits de anchura por la que llegan las muestras.
- Una entrada input_enable que indica que el dato colocado en input es válido en el ciclo actual.
- Una salida de datos output, y un indicador output_empty que indica a la baja la presencia de datos. Estos serán colocados en output el ciclo siguiente a levantar la entrada output_read.
- Una salida busy, que comienza a **1** y baja tras emitirse el último byte por output.

Los controles de entrada y salida se hacen mediante las máquinas de estados que se muestran en los Algoritmos 18 y 19.

El control de ejecución es totalmente combinacional. El EBCoder está activo siempre que las colas de salida del mismo tengan hueco, ya que guardan tanto los bytes como los bits de enable todos en una misma palabra de 27 bits. Por otro lado, los datos de salida del EBCoder se inyectan en las colas siempre y cuando exista un bit de enable distinto de cero. Para evitar que el mismo dato se inserte varias veces, se mira además que tanto las señales de actividad como de enable del EBCoder estén a **1**.

7.11. Configuración y limitaciones

Existen una serie de opciones a la hora de instanciar el módulo:

La primera opción es el tamaño de las colas de entrada y salida. Como veremos en el Capítulo 8, la cola de entrada no es problemática en cuanto a tamaño pues su mayor utilidad es servir de interfaz de entrada fácil de manejar. Un tamaño pequeño será suficiente. La cola de salida servirá de búfer del resultado, pero un tamaño pequeño tampoco impedirá el correcto funcionamiento, ya que si se llena, el módulo será capaz de detener su ejecución sin problemas.

Una opción que queda desactivada en el empaquetado es la de ajustar la profundidad de bits. El EBCoder sí la permite, pero al empaquetar queremos una interfaz de comunicación a nivel de byte, y profundidades que no fueran múltiplo de un byte serían complejas de tratar. En cualquier caso, si se tuvieran que codificar datos de menor profundidad se pueden rellenar los planos extra de bits con ceros, que serán codificados con el codificador de carrera de forma muy eficiente, añadiendo uno o dos bytes por bloque, un sobrecoste asumible.

La última opción es la de ajustar el tamaño de bloque. Si bien el estándar permite cualquier tamaño, aquí, por decisiones de diseño, estamos limitados a que el número de filas sea múltiplo

Algoritmo 18: Control de entrada al EBCoder

```
1  switch state_current do
2      case IDLE do
3          if hay datos en la cola de entrada then
4              state_next ← READ_ONE;
5              Activa lectura en la cola de entrada;
6          end
7      end
8      case READ_ONE do
9          data_next [15:8] ← salida de la cola de entrada;
10         if hay datos en la cola de entrada then
11             state_next ← READ_ONE;
12             Activa lectura en la cola de entrada;
13         end
14     end
15     case READ_TWO do
16         data_next [7:0] ← salida de la cola de entrada;
17         state_next ← SAVING;
18     end
19     case SAVING do
20         Activar la entrada del ebcoder y enviar data_curr;
21         state_next ← IDLE;
22     end
23 end
```

Algoritmo 19: Control de salida del EBCoder

```
1  switch state_current do
2      case IDLE do
3          if hay elementos en las colas de salida del EBCoder then
4              Activar la lectura de las colas;
5              state_next ← OUTPUT_ONE;
6          end
7      end
8      case OUTPUT_ONE do
9          if el primer byte es válido then
10             if la cola de salida no está llena then
11                 Activar escritura en la cola de salida;
12                 El primer byte es la entrada de la cola de salida;
13                 state_next ← OUTPUT_TWO;
14             end
15         else
16             state_next ← OUTPUT_TWO;
17         end
18     end
19     Los casos para OUTPUT_TWO y OUTPUT_THREE son idénticos a OUTPUT_ONE,
         cambiando el siguiente estado por OUTPUT_THREE y IDLE respectivamente.
20 end
```

de 4. La estructura de la memoria y las generaciones de vecindarios asumen que la distribución de todos los datos sigue el recorrido en zig-zag para tiras de 4 muestras de alto. Si el bloque no tuviera la altura adecuada, la última tira no tendría 4 muestras, y el circuito no trabajaría con los resultados adecuados.

En caso de necesitarse un bloque de tamaño menor, se puede ampliar hasta tener altura múltiplo de 4 con ceros. Esto solo es necesario para los bloques que tocan con el borde inferior de la imagen, ya que los demás se pueden tomar siempre con la altura requerida, por tanto el impacto será pequeño.

En anchura sí se puede escoger cualquier tamaño sin restricciones, y de hecho el tamaño total de bloque se podría ampliar arbitrariamente por encima de 64×64, siempre y cuando la FPGA disponga de memoria suficiente.

Capítulo 8

Probando el EBCoder para FPGA

Teniendo listo el módulo codificador, el siguiente paso es crear un test para probarlo. Las pruebas se realizarán, en primer lugar, de forma teórica en simulación. Una vez funcionen, se probarán directamente sobre FPGA.

8.1. Metodología de pruebas

A estas alturas disponemos de un software (Capítulo 5) de compresión, que correctamente comprime y descomprime imágenes completas. Utilizaremos el módulo de codificación de bloques para generar varios casos de prueba, que serán contrastados con la implentación hardware (Figura 8.1).

Cada caso de prueba consiste en un archivo de entrada que representa los datos en crudo, y otro de salida que representa los datos codificados, incluyendo el marcador de terminación `0xfffe`.

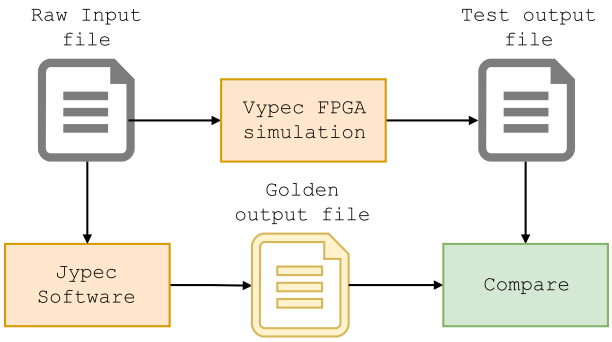


Figura 8.1: Flujo de las pruebas en simulación. Los datos generados se prueban en ambas implementaciones y se comparan posteriormente.

- La entrada contiene, en el orden de recorrido en zig-zag del estándar, todas las muestras del bloque a probar, con profundidad de 16 bits. El orden es importante pues tanto el software como el hardware asumen dicha ordenación en sus respectivos lectores de archivo.
- La salida se genera desde Jypec, que ya está probado, y se considera un archivo *golden*. La salida de la implementación hardware deberá coincidir con este archivo para considerarse correcta (salvo fallo de implementación en Jypec, por supuesto).

El archivo en crudo se genera pseudoaleatoriamente. Para ello se elige un número al azar (idealmente coprimo con 2^{16} para conseguir una secuencia sin repeticiones) y se colocan en sucesión múltiplos sucesivos módulo 2^{16}, como muestra la Figura 8.2. También se prueba con algunos ficheros concretos de casos extremos (todas las muestras a cero, a valor máximo $(2^{15}-1)$, a valor mínimo $(-2^{15}+1)$).

El bit más significativo es utilizado para el signo, mientras que el resto representarán la magnitud.

El método de generación se basa en el hecho de que la secuencia:

$$S_{a,b} = \{a * n \mod b, n \in \mathbb{N}\} \tag{8.1}$$

Tiene un periodo igual a $b/mcd(a,b)$. Por tanto, de ser a y b coprimos, recorrerá todos los números $n \in [0,b)$ antes de comenzar a repetirlos. Esto es interesante pues aseguramos que se prueba el módulo con números distintos, pero generables de una manera fácil y predecible, para poder repetir una ejecución en caso de fallo, y detectar de dónde viene el error.

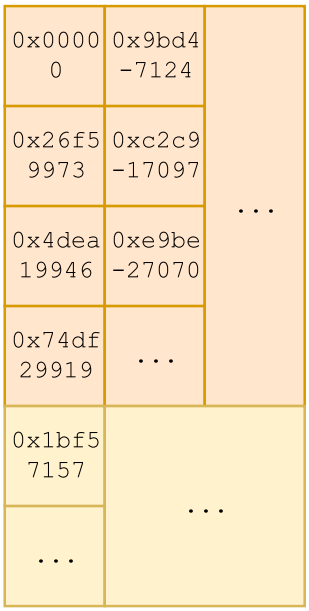


Figura 8.2: Ejemplo de generación de un bloque pseudoaleatorio de 64 × 64 partiendo del número primo 9973. Como el archivo sigue un orden de zig-zag, los múltiplos no son consecutivos por filas o columnas.

8.2. Obteniendo resultados de la implementación

En la Sección 7.10 habíamos dejado el EBCoder dentro de un módulo que gestionaba entrada y salida a nivel de byte. Para simplificar la fase de pruebas, vamos a probar el EBCoder por sí solo, y después probaremos el módulo empaquetado con las interfaces a nivel de byte.

8.2.1. Comprobando la corrección

El primer test contiene dos procesos. Por un lado se controlan las entradas al módulo (Algoritmo 20), inyectando las muestras de entrada, y por otro se van leyendo las salidas del módulo (Algoritmo 21), que se guardan en un archivo.

Algoritmo 20: Proceso de entrada

```
1  reset ← 1;
2  esperar un ciclo de reloj;
3  reset ← 0;
4  clock_enable ← 1;
5  esperar un ciclo de reloj;
6  while aún hay datos do
7      data_in ← siguiente dato en la secuencia;
8      data_in_enable ← 1;
9      esperar un ciclo de reloj;
10     data_in_enable ← 0;
11     esperar un número variable de ciclos de reloj;
12 end
13 esperar indefinidamente;
```

El proceso de entrada activa el módulo tras hacer una secuencia de reseteo del mismo. A continuación toma los datos (bien de archivo o generados pseudoaleatoriamente siguiendo la Ecuación (8.1)), los coloca en la entrada y los graba en memoria levantando un ciclo la señal de dato activo. Antes de pasar al siguiente dato se puede esperar un número variable de ciclos para probar el comportamiento del circuito ante diferentes latencias de datos.

El proceso de salida espera a que comience a funcionar el módulo, y una vez lo ha hecho

Algoritmo 21: Proceso de salida

```
1  esperar a que el reset esté activo por un ciclo;
2  esperar a que se levante busy;
3  while busy = 1 do
4      for cada pareja (byte, valid) en la salida del EBCoder do
5          if valid = 1 y clock_enable = 1 then
6              escribir byte en el archivo de salida;
7          end
8      end
9      esperar un ciclo de reloj;
10 end
11 esperar indefinidamente;
```

comienza a observar la salida del codificador. Cada vez que haya un byte válido, lo tomará y escribirá en el archivo de salida.

Una vez obtenido el resultado, comprobamos mediante un editor hexadecimal ambos archivos. En este caso se ha utilizado HxD [77]. HxD permite comparar byte a byte los archivos (Figura 8.3), y decirnos exactamente en cuál difieren. Posteriormente podemos ir al simulador e inspeccionar las señales en ese instante de tiempo, para ver qué ha podido fallar.

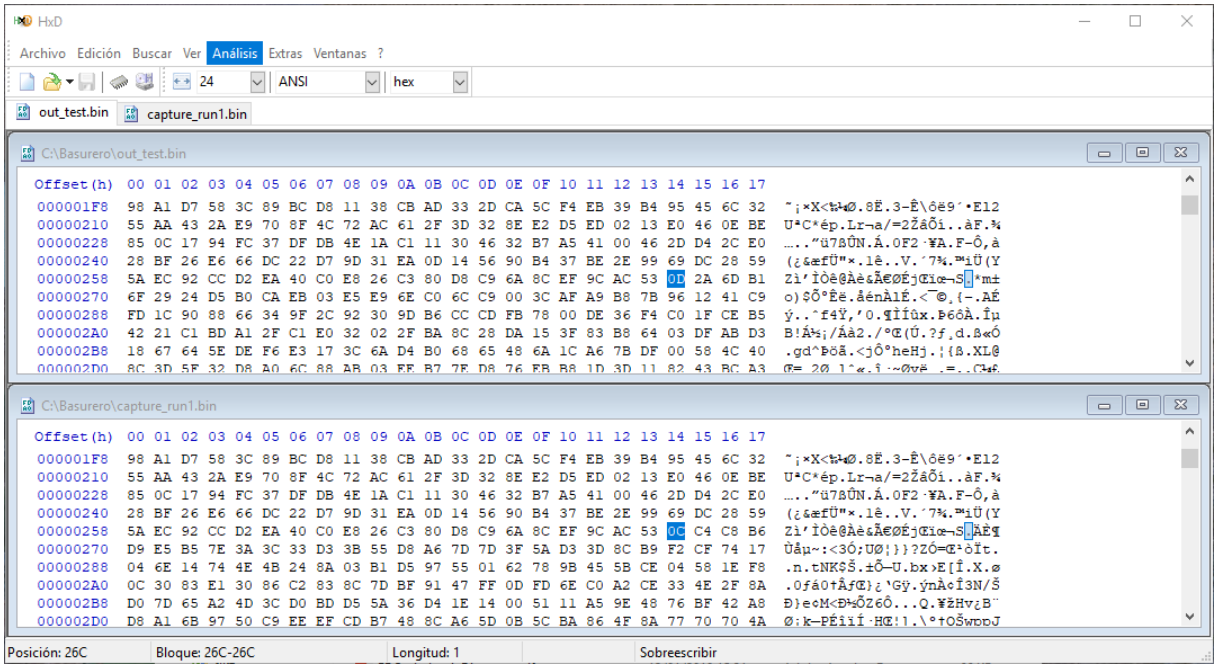


Figura 8.3: Ejemplo de análisis con HxD. En este caso la salida coincide hasta cierto punto (donde falló la implementación en FPGA) que queda señalado tras el análisis. El número de byte queda indicado en pantalla.

8.2.2. Probando el empaquetado

Una vez visto que las salidas de los tests funcionan es momento de probar el EBCoder con las interfaces de byte. Aquí introducimos unas colas de entrada y de salida. La corrección de las colas ya está probada [78], pero el sistema de control que las conecta con el EBCoder no. Para ello, el test forzará los casos extremos (colas llenas o vacías) a fin de explorar el comportamiento del diseño ante diferentes situaciones.

El proceso de envío, que vemos en el Algoritmo 22, cambia ligeramente. Ahora enviamos las muestras a nivel de byte, por lo que tenemos que dividirlas en dos trozos. En la linea 11 esperamos un número variable de ciclos. La idea es probar qué pasa si se llena la cola o si no hay datos:

Al llenarse perdemos datos, ya que el test no controla la activación de la señal de entrada llena. Es importante adelantar que, como posteriormente utilizaremos una comunicación UART con el módulo, tampoco va a ser posible saber cuándo está llena. Por tanto detectar este fallo nos indica que no debemos enviar datos más rápido de lo que se pueden extraer de la cola de entrada. En un protocolo de comunicación más sofisticado sí se podría tener en cuenta esa señal de llenado para parar el envío hasta que se vacíe.

En cualquier caso, tampoco nos supondrá un problema a la hora de realizar pruebas, ya que el módulo alcanzará una velocidad de lectura varios órdenes de magnitud superior al que se puede conseguir en escritura por UART.

En cuanto a tener una cola vacía, tampoco será un problema. El módulo simplemente para hasta que haya datos disponibles.

Algoritmo 22: Proceso de entrada

```
1  reset ← 1;
2  esperar un ciclo de reloj;
3  reset ← 0;
4  while aún hay datos do
5      fifo_in_data ← primer byte de la entrada;
6      fifo_in_data_enable ← 1;
7      esperar un ciclo de reloj;
8      fifo_in_data ← segundo byte de la entrada;
9      esperar un ciclo de reloj;
10     fifo_in_data_enable ← 0;
11     esperar un número variable de ciclos de reloj;
12 end
13 esperar indefinidamente;
```

El proceso de salida también cambia, como vemos en el Algoritmo 23. Tras esperar a que se active el módulo, esperamos de nuevo en la línea 2 para hacer que se llene o no la cola de salida. Posteriormente, vamos leyendo datos mientras los tengamos disponibles, hasta que se acaben. Entre lecturas también podemos esperar un número variable de ciclos, para hacer que la cola nunca se vacíe del todo (línea 9).

En este caso comprobamos que, aunque se llene la cola de salida, los mecanismos diseñados para bloquear la ejecución funcionan adecuadamente. Además no existen interferencias con la cola de entrada, ya que las lecturas van aparte, y se pueden hacer todas aunque la cola de salida se bloquee desde el principio.

8.3. Preparaciones finales

Tras haber comprobado que las simulaciones funcionaban como esperábamos, es el momento de probar el circuito en la FPGA. Tenemos el empaquetado con las interfaces de anchura 1 byte de entrada y salida. Para comunicarlo con el ordenador, vamos a colocarle un módulo *Universal Asynchronous Receiver-Transmitter* UART para comunicarnos por puerto serie.

El UART en este caso manejará únicamente dos cables: Rx y Tx, respectivamente de recepción

Algoritmo 23: Proceso de salida

```
1  esperar a que busy = 1;
2  esperar un número variable de ciclos de reloj;
3  while busy = 1 o fifo_out_empty = 0 do
4      if fifo_out_empty = 0 then
5          fifo_out_read_enable ← 1;
6          esperar un ciclo de reloj;
7          fifo_out_read_enable ← 0;
8          fifo_out_data contiene el dato leído;
9          esperar un número variable de ciclos de reloj;
10     end
11 end
12 esperar indefinidamente;
```

y de transmisión de datos.

Cada uno de ellos se controla por separado con los módulos de recepción y transmisión del UART (tomados de [79]), que se han unido con el nuestro. Las salidas de dato y activación de la entrada del UART se han conectado directamente a la cola de entrada. Para la salida, se ha diseñado un módulo que hace de interfaz entre la cola de salida de nuestro módulo con las señales de control del UART, como puede verse en el Algoritmo 24.

Algoritmo 24: Control de salida del UART

```
1  fifo_out_read_enable ← 0;
2  state_next ← state_current;
3  uart_send_request ← 0;
4  switch state_current do
5      case IDLE do
6          if hay datos en la cola de salida then
7              fifo_out_read_enable ← 1;
8              state_next ← OUT_READY;
9          end
10     end
11     case OUT_READY do
12         if uart_busy = 0 then
13             uart_send_request ← 1;
14             state_next ← UART_SENDING;
15         end
16     end
17     case UART_SENDING do
18         if uart_done = 1 then
19             state_next ← IDLE;
20         end
21     end
22 end
```

Solo queda una última cosa que ajustar antes de las pruebas físicas: el ciclo de reloj. La simulación no tiene en cuenta los tiempos de propagación de las señales, y ejecuta todo en cuanto viene el flanco de reloj. En la realidad no es así, y un reloj mal configurado puede hacer que nuestro circuito no de la respuesta adecuada.

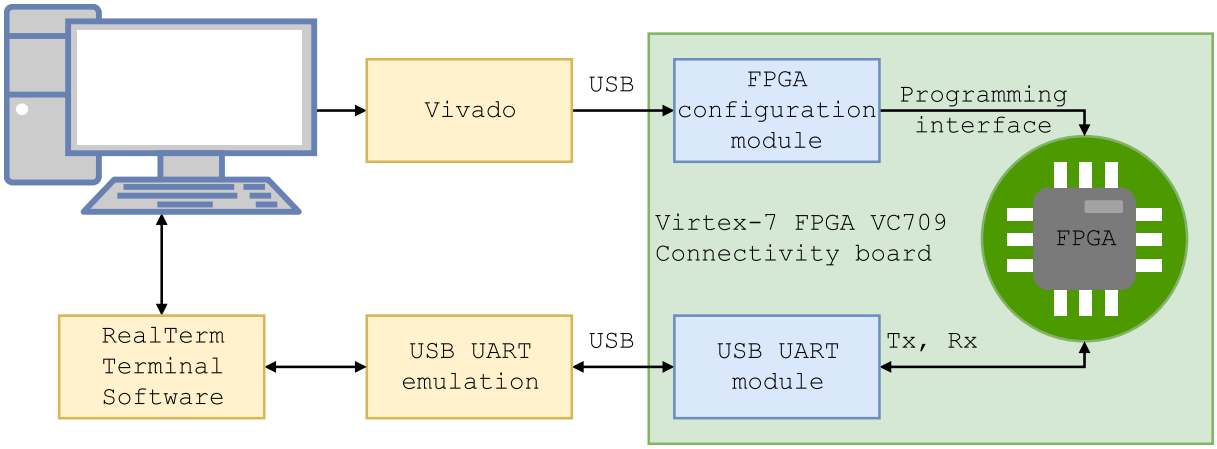


Figura 8.4: Los diferentes elementos utilizados para el testing sobre placa para la Virtex 7. El ordenador utiliza Vivado y RealTerm para la comunicación con la FPGA, mediante dos cables USB.

Así pues, tras obtener los resultados teóricos de implementación sobre el ciclo de reloj (ver Tabla 8.3), colocamos un divisor de frecuencia que la reduce a $50MHz$, más que suficiente para procesar los datos, por muy rápido que vengan por la UART.

8.4. Pruebas físicas

Los módulos transmisor y receptor del UART ya están probados ([79]), al igual que la corrección de nuestro EBCoder, por tanto ahora solo queda pasar a placa. Las pruebas se han realizado sobre la placa VC709 de Xilinx.

Los datos de entrada se envían desde un ordenador, que es el mismo que lee las salidas, para posteriormente comprobar su corrección. Las conexiones entre los diferentes elementos involucrados en el testing pueden verse en la Figura 8.4. Dichos elementos son:

- **Vivado** [80]: Es el IDE desde el cual se realiza el flujo entero de compilación, desde la síntesis hasta la generación del archivo de configuración de FPGA. El propio entorno gestiona el envío del archivo de configuración mediante el "gestor de hardware". Basta conectar el USB de configuración al ordenador, encender la placa, y el entorno automáticamente la detectará y ofrecerá la posibilidad de configurar.

- **Realterm** [81]: Realterm permite la interacción fácil con numerosos protocolos de puerto serie, entre los que se incluye UART. Dado que el ordenador no tiene un puerto UART físico, el puerto se emula mediante USB. La placa VC709 también cuenta con un puerto USB, y traduce las señales a los dos cables Rx y Tx propios del UART.

  Realterm permite el envío de archivos binarios byte a byte, con selección de la velocidad de envío. Además da la opción de capturar los bytes recibidos y guardarlos en un archivo, lo cual será de utilidad para poder compararlos con el golden posteriormente.

- **Virtex VC709 Connectivity Kit** [82]: Este kit incluye el chip de FPGA integrado en una placa, junto con numerosos elementos de interfaz con la misma. Entre ellos dos puertos USB que nos servirán tanto para configurar la placa, como para hacer la comunicación UART con el ordenador.

  De cara a la configuración no nos preocupamos de nada, todo viene hecho. De cara al UART, deberemos configurar el archivo de restricciones del proyecto para atar los pines Rx y Tx a la FPGA en los pines AU33 y AU36 respectivamente.

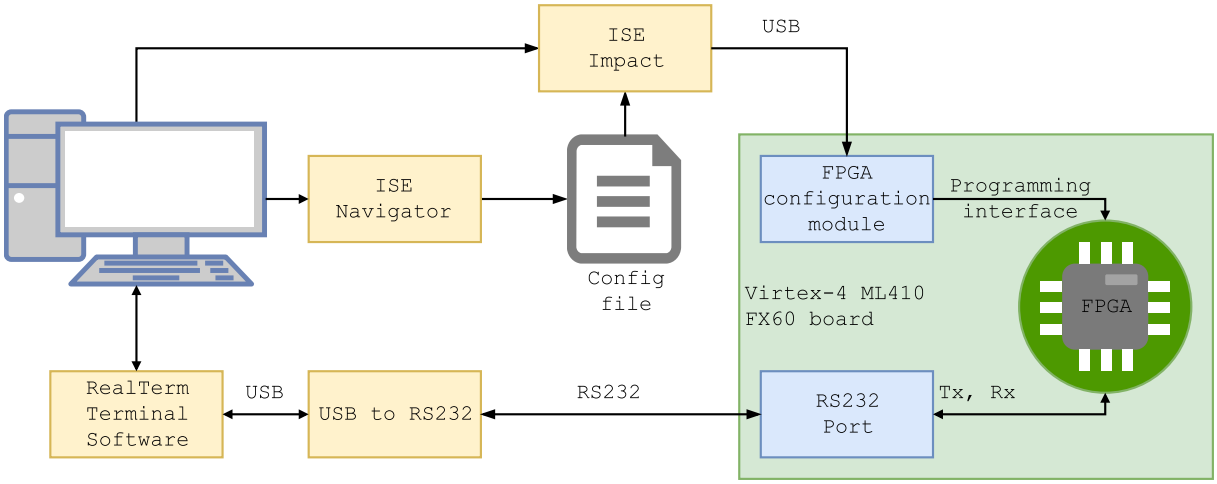


Figura 8.5: Los diferentes elementos utilizados para el testing sobre placa para la Virtex 4. El ordenador utiliza ISE Navigator para la generación de archivos de configuración. RealTerm para la comunicación con la FPGA directamente mediante RS232. Impact para la configuración de la placa con el archivo generado por ISE.

Modelo	Slices	LUT	FF	DSP	18k RAM	IO
Virtex 4 (xc4vfx60-11-ff1152) [85]	25280	50560	50560	128	232	576
Virtex 7 (xc7vx690t-2-ffg1761) [86]	108300	433200	866400	3600	2940	1000

Tabla 8.1: Recursos disponibles en las FPGAs utilizadas en las pruebas.

Para estas pruebas se han utilizado los mismos conjuntos de datos definidos en la Sección 8.1, y se ha comprobado satisfactoriamente la coincidencia de los resultados obtenidos en software y hardware.

Por otra parte se han realizado pruebas con la placa ML410 FX60, que contiene una Virtex 4. La razón es la existencia de una fpga equivalente a la Virtex 4 pero resistente a radiación. Dado que algunas imágenes hiperespectrales se toman desde satélites, es interesante ver la capacidad de compresión en entornos hostiles. La conexión con dicha placa se aprecia en la Figura 8.5

Además de Realterm, utilizado ya para la comunicación con la Virtex 7, se utilizan para la Virtex 4 los siguientes programas:

- **Ise navigator** [83]: El IDE desde donde se escribe y compila el código pertinente, equivalente al vivado para la Virtex 7.
- **Impact** [83]: Parte de la suite de ISE, se utiliza para configurar la FPGA.
- **Virtex ML410 FX60** [84]: Placa que equipa a una Virtex 4 con los conectores necesarios para su configuración, depuración y comunicación.

8.5. Rendimiento del módulo

Para ver el rendimiento se han utilizado dos placas. La ya mencionada Virtex 7 del VC709 Connectivity Kit, y una Virtex 4 de la placa ML410 FX60.

Veamos en primer lugar las características de ambas FPGAs en la Tabla 8.1.

Pasamos a los resultados de ocupación y consumo aproximado, tanto para el EBCoder "desnudo" de la Sección 7.8 como para el módulo con interfaces de comunicación mediante colas de la Sección 7.10, ambos en la Tabla 8.2.

Modelo	Slices	LUT	FF	DSP	18k RAM	IO	Consumo
V4	11571	14780	481	0	0	57	707,46mW
V4-wrapper	11735	15035	558	0	0	31	691,08mW
V7	n/a	1185	428	0	7	57	323,99mW
V7-wrapper	n/a	1328	530	0	7	31	324,01mW

Tabla 8.2: Ocupación del EBCoder y del EBCoder con interfaz sobre ambas placas probadas. También se muestra el consumo estimado.

Modelo	V4	V4-wrapper	V7	V7-wrapper
Frecuencia máxima	31,707MHz	32,057MHz	62,763MHz	62,764MHz

Tabla 8.3: Frecuencia máxima para las diferentes configuraciones y FPGAs.

Lo primero que observamos es la gran diferencia en número de LUTs entre las dos placas. El sintetizador para la Virtex 4 (Ise 2013.10.13) detecta las memorias RAM, pero no es capaz de introducirlas en los bloques por limitaciones de la placa, con lo que las implementa sobre las LUT. El sintetizador de la Virtex 7 (Vivado 2016.3) sí es capaz de ponerlas en su sitio, utilizando 7 bloques de RAM en lugar de algo más de 10000 LUTs, con lo que la ocupación baja notablemente.

Los DSP no se utilizan para nada, debido a no existir operaciones complejas como multiplicaciones. Las entradas y salidas son, como es de esperar, iguales en ambas placas, y el consumo es algo más del doble en la Virtex 4, cuya estimación es sorprendentemente más baja en la versión con interfaz de colas. (En cualquier caso, estas estimaciones sirven meramente como aproximación).

Ambas placas son pues capaces de albergar el algoritmo sin dificultades, y queda ver qué velocidad de ejecución consiguen. La frecuencia máxima reportada por los sintetizadores se observa en la Tabla 8.3.

La frecuencia es prácticamente el doble en la Virtex 7, y observamos que la adición de la interfaz no supone pérdidas en la frecuencia máxima, por lo que el cuello de botella estaría en el propio codificador.

Ahora bien, ¿Qué significan estas frecuencias?. Para medir la velocidad, vamos a tener en cuenta no la de salida (que puede ser variable según el bloque) sino la de entrada (que será constante). Supongamos un bloque de tamaño $x \times y$ con una profundidad de bits p.

A la hora de codificarlo, se van a dar p pasadas de limpieza y $p-1$ pasadas de refinamiento y significancia. Como el signo se codifica aparte en una de las tres pasadas anteriores, podemos asumir una pasada de codificación de signo (esto es en un caso peor, ya que no siempre se codifican todos los signos) que tardará $x \cdot y$ ciclos. Las pasadas de refinamiento y significancia siempre van a tardar un ciclo por bit, por lo que añaden $2(p-1)xy$ ciclos. La pasada de limpieza, en el caso peor, tendrá una rotura de codificación de carrera por cada cuatro muestras, lo cual suma 2 ciclos extra, o 0,5 por muestra. En total demorará la ejecución $1{,}5pxy$ ciclos.

Obviamente esto es un caso peor propuesto como cota superior. Un análisis más detallado probablemente nos diera una cota superior más pequeña, ya que por ejemplo no es posible que la pasada de limpieza siempre genere roturas de codificación en carrera. En cualquier caso, el total asciende a:

$$xy + 2(p-1)xy + 1{,}5pxy = xy(3{,}5p - 1) \approx 3{,}5pxy \tag{8.2}$$

El caso mejor se calcula de manera similar, ahora asumiendo que el signo no se codifica nunca, y que la limpieza se hace sin interrupciones. En total:

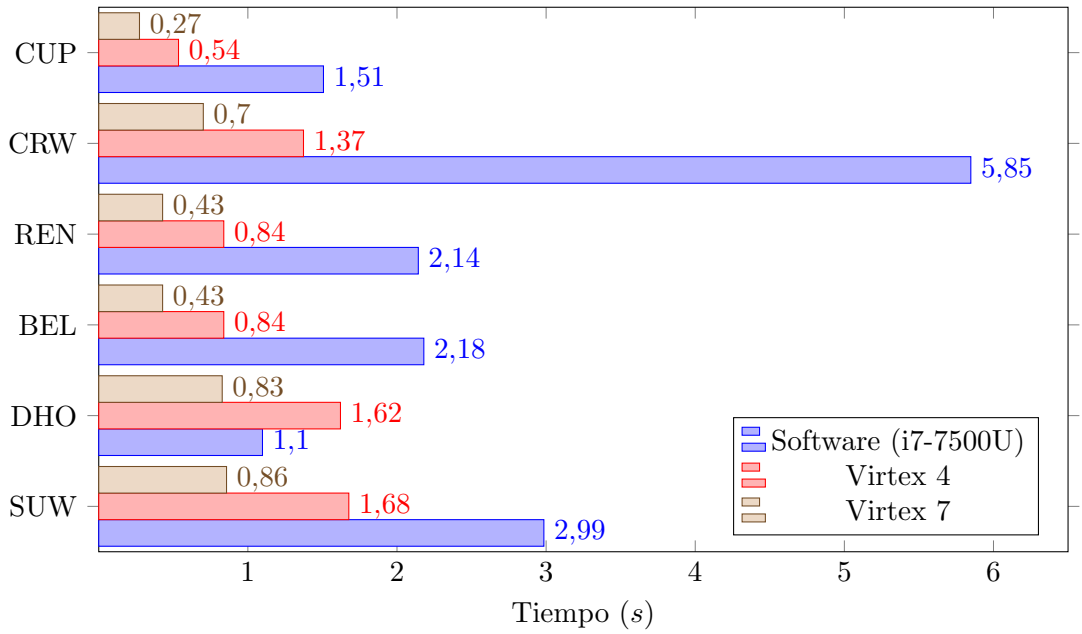


Figura 8.6: Comparativa entre el rendimiento software y FPGA para el EBCoder sobre diferentes imágenes.

$$2(p-1)xy + pxy = xy(3p-2) \approx 3pxy \tag{8.3}$$

Por tanto, el tiempo de ejecución t se situará entre $3pxy \leq t \leq 3{,}5pxy$. En todos nuestros cálculos posteriores utilizaremos el caso peor $3{,}5pxy$ como referencia.

8.5.1. Comparativa con software

¿Qué supone esto?. Imaginemos que estamos comprimiendo la imagen CRW de dimensiones $224 \times 614 \times 512$ con los parámetros por defecto, es decir, profundidad de 10 bits y reducción a 4 dimensiones. En este caso $x = 614$, $y = 512$ y $p = 10$. Además tenemos que repetir este proceso 4 veces, una para cada dimensión. En total gastaremos:

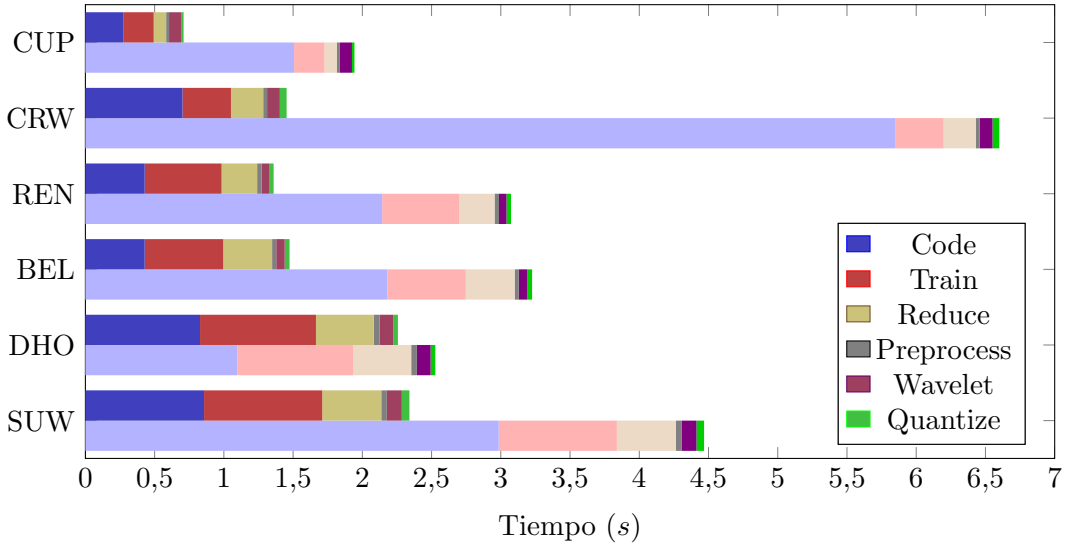
$$3{,}5 \cdot 10 \cdot 614 \cdot 512 \cdot 4 = 44011520 \text{ ciclos}$$

Lo que supone 1,3729 segundos en la Virtex 4, y tan solo 0,7012 segundos en la Virtex 7. Si lo comparamos con los 5,848 segundos que tardaba la codificación software, tenemos un speedup de 4,26× y 8,33× respectivamente, algo nada desdeñable. En la Figura 8.6 podemos ver gráficamente el mismo cálculo comparado con los resultados de la Figura 6.16.

Es claro ver que por norma general, la implementación hardware en ambas FPGAs es mejor que la software. Solo en DHO mejora el software a la Virtex 4, debido a que dicha imagen tiene valores muy bajos en sus píxeles, y el software funciona notablemente más rápido en la codificación de carrera.

Por lo demás, se obtiene un speedup medio de unas 3× para la Virtex 4, y de 6× para la Virtex 7. Podemos ver el impacto sobre el coste total de la ejecución del algoritmo en la Figura 8.7.

El speedup ronda los 2× de media para las imágenes de prueba, consiguiendo mejora en todas. El cuello de botella, que era la codificación, ha desaparecido casi en su totalidad, siendo interesante para un futuro investigar la aceleración de otras partes del algoritmo.



Speedup	SUW	DHO	BEL	REN	CRW	CUP
Codificación por bloques	3,483×	1,326×	5,090×	5,003×	8,340×	5,516×
Proceso completo	1,875×	1,115×	2,121×	2,198×	2,868×	2,628×

Figura 8.7: Comparación para cada imagen del tiempo total acelerando (color fuerte) la codificación con FPGA (Virtex 7) vs no acelerándola (colores claros). Vemos que la codificación ya no es el cuello de botella, y se equipara con otras fases.

w\Imagen	SUW	DHO	BEL	REN	CRW	CUP
3	107	107	61	61	86	43
2	106	106	62	62	84	43
1	104	104	56	56	80	36
0	95	95	50	50	80	36

Figura 8.8: Número de bloques en que se divide cada imagen para diferentes valores de **`-w`**.

8.6. Mejoras con paralelización

La Virtex 4 andaba justa de espacio, pero la Virtex 7 iba bien holgada con tan solo un 1 % de ocupación.

Recordamos de la Sección 4.2.4 que la codificación de los bloques es totalmente independiente. Con el diseño actual estamos instanciando un único módulo de EBCoder que se encarga de codificar todos los bloques de la imagen. ¿Y si ponemos varios a trabajar en paralelo?.

En la Figura 8.8 vemos en cuantos bloques se está dividiendo nuestra imagen. El número de bloques en que se divide una imagen depende de sus dimensiones y de las pasadas de ondícula que se den, pues a cada bloque solo pueden pertenecer muestras de la misma subdivisión de ondícula.

En un primer momento podríamos pensar en instanciar tantos codificadores como bloques tiene la imagen. Si bien esto es posible, se puede mejorar aún más, pues si la reducción dimensional baja a m dimensiones, cada una de ellas se puede codificar independientemente, por tanto podríamos paralelizar en $m \cdot$ numbloques módulos diferentes.

En la Virtex 7 cabrían aproximadamente 200 codificadores simultáneamente, previo diseño de un módulo gestor que fuera capaz de alimentarlos a todos de datos a la velocidad necesaria

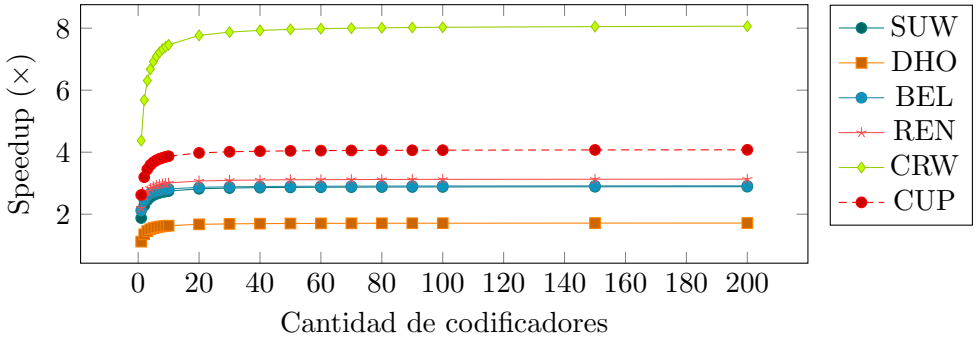


Imagen	SUW	DHO	BEL	REN	CRW	CUP
Límite teórico	2,90×	1,72×	2,92×	3,14×	8,10×	4,09×

Figura 8.9: Resultados teóricos de speedup sobre el proceso completo de compresión con diferente número de codificadores trabajando simultáneamente. El límite teórico es para infinitos codificadores trabajando simultáneamente.

para su consumo. Bajo este supuesto, la mejora de la codificación sería de 200× sobre la ya conseguida anteriormente. Con esto prácticamente todo el tiempo quedaría dedicado a las demás fases, acercándonos asintóticamente al límite teórico mostrado en la Figura 8.9.

Como vemos, la mayor mejora se consigue de 1 a 3 codificadores. A partir de ahí el speedup no aumenta mucho debido a las partes del algoritmo sin acelerar. Añadir más de 10 codificadores no mejoraría mucho ya que el límite teórico es prácticamente alcanzado con 10. Recordemos que el límite de codificadores viene dado por la cantidad de bloques, así que poner entre 1 y 10 codificadores no sería problema para ninguna imagen (Figura 8.8).

8.7. Rendimiento en tiempo real

Es muy interesante saber si hemos conseguido alcanzar el tiempo real, ya que esto supondría la posibilidad de comprimir las imágenes según son tomadas por el sensor. La velocidad de referencia que tomamos es la de captura de uno de los sensores más nuevos, el AVIRIS-NG, que se sitúa en 74MB/s [15]. Vemos los resultados en la Figura 8.10.

Es claro que la aproximación inicial se alejaba demasiado del tiempo real, siendo hasta casi 20 veces mayor en algunos casos. La mejora del entrenamiento ($t = 0{,}01$) acerca notablemente el tiempo de ejecución al de captura, que es finalmente superado al introducir la opción híbrida con coprocesamiento de la codificación de bloques en FPGA.

Las imágenes más grandes (SUW y DHO) son las que más mejora presentan, por lo que el crecimiento de la resolución y tamaño de las imágenes hiperespectrales, con el desarrollo de nuevos sensores, no es un problema para este algoritmo.

Incluso la FPGA endurecida para el espacio (V4) es capaz de conseguir la compresión en tiempo real para las imágenes más grandes. El hecho de existir modelos más nuevos, como la Virtex 5 QV [87], con más prestaciones, posibilita la compresión de todo tipo de imágenes incluso en condiciones hostiles, partiendo del supuesto de que el resto del algoritmo, que se hace sobre CPU, también estuviera endurecido.

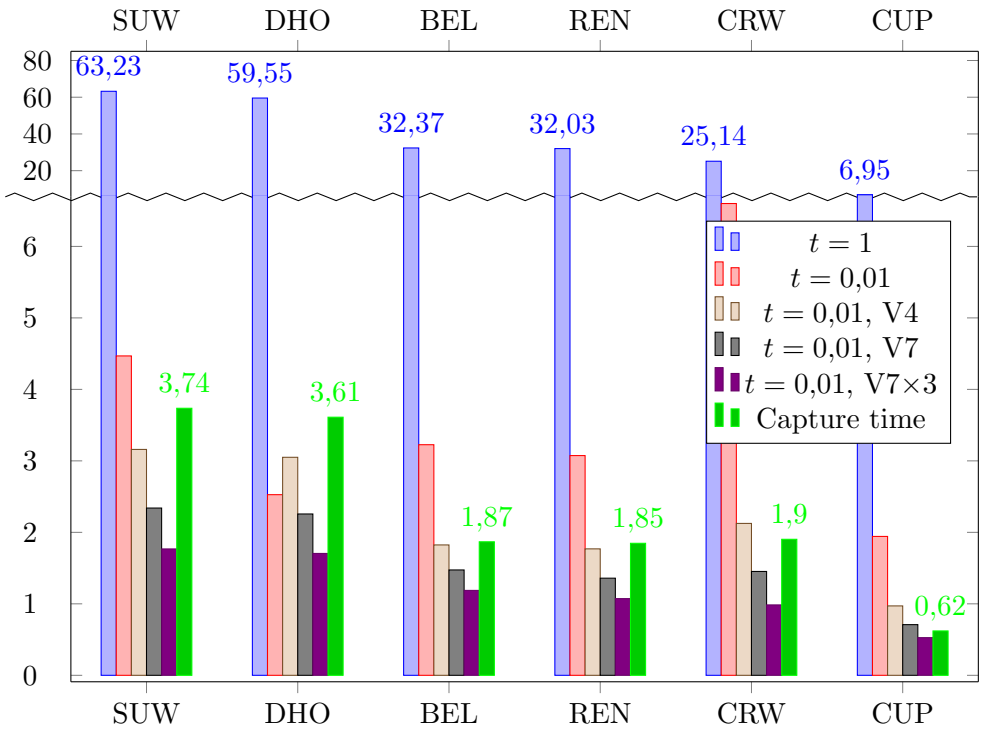


Figura 8.10: Resultados de tiempo (en segundos) de la ejecución del algoritmo a lo largo de las diferentes mejoras.

8.8. Comparativa con otras implementaciones

Es bien sabido en la literatura que el mayor cuello de botella en JPEG2000 es la codificación [88-92], por lo que numerosas arquitecturas han sido diseñadas para paliar el problema, especialmente orientadas a mejoras en el codificador MQ.

Como referencia, la implementación desarrollada en este trabajo (codificador MQ + EB) funciona a una velocidad de 62,76MHz, con el codificador MQ funcionando por sí solo a 78,75Mhz. Se consigue así una velocidad máxima de 62.76MS/s (mega símbolos por segundo) para el EBCoder, y de 78,75MS/s en el MQCoder, al ser capaz de codificar cada bit del bloque de datos en exactamente un ciclo en condiciones idóneas.

En [88] consiguen una velocidad de 137,7MS/s sobre una FPGA Stratix, haciendo una estructura segmentada del codificador MQ que mejora la etapa de renormalización de los registros. No ofrecen integración con ninguna arquitectura del EBCoder, pero es de asumir que aplicar estrategias similares en un futuro podría mejorar el rendimiento del circuito aquí desarrollado.

Otra mejora diferente surge en [89], donde mediante un procesamiento paralelo, que procesa las parejas (bit, contexto) de dos en dos, consiguen velocidades de 96,6MS/s pese a aumentar el ciclo de reloj. De nuevo una idea que de integrarse junto al EBCoder podría mejorar muy seguramente su rendimiento. Una arquitectura similar, también en FPGA es presentada en [90], mostrando sin embargo un rendimiento inferior de 60MS/s.

Por último, existen implementaciones directamente sobre chips CMOS [91, 92] del codificador entero (MQ + EB) que alcanzan hasta 142MS/s. Es un paso más después del prototipado en FPGA, y sin duda cualquier implementación sobre silicio "duro" frente al "blando" de las FPGAs mejorará los resultados obtenidos con estas últimas.

Capítulo 9

Conclusiones

La mayor utilidad de las imágenes hiperespectrales radica en poder realizar estudios a distancia en numerosas disciplinas, sin tener que interactuar con lo que se estudia a nivel físico, más allá de, quizá, iluminarlo.

Los datos recogidos indican la cantidad de luz recibida en cada pixel de la imagen. Además de la luz visible, se recoge información de otras bandas del espectro, como infrarroja y ultravioleta. Todos estos datos, que permiten obtener una gran cantidad de información desde una simple imagen, hacen también que su tamaño sea notable.

La compresión de grandes conjuntos de datos es en muchas ocasiones necesaria por restricciones de almacenamiento. Sobre todo en el ámbito científico, donde a menudo es necesario recopilar gran cantidad de muestras antes de poder hacer un análisis en profundidad de todo el conjunto.

A veces las restricciones de espacio no vienen dadas por el almacenamiento principal, sino por la memoria intermedia entre el sensor que las captura y dicho almacenamiento. Tener ahí un cuello de botella puede suponer que el sensor tenga que estar ocioso mientras se vuelcan los datos, y perdamos tiempo u oportunidad de capturar nueva información.

Pero claro, tampoco es útil una buena compresión si no la hacemos con rapidez, porque entonces surgirán nuevos cuellos de botella.

Las FPGAs son un buen aliado para ayudar en estos problemas. Chips reconfigurables con los que poder acelerar los algoritmos de compresión hasta alcanzar rendimientos en tiempo real, pudiendo colocarse directamente como interfaz entre sensor y memoria, eliminando los problemas mencionados de un golpe.

En este trabajo hemos comenzado analizando las técnicas de compresión más utilizadas, desde los comienzos de la teoría de la información hasta los algoritmos más potentes como JPEG2000. Hemos visto sus aplicaciones para la compresión de imágenes hiperespectrales, encontrando buenos resultados en la literatura para la mezcla de reducción dimensional con algoritmos tradicionales de compresión de imágenes no hiperespectrales.

Partiendo de esa idea, se han aplicado variaciones al algoritmo, introduciendo la posibilidad de utilizar numerosos reductores dimensionales diferentes, detección de anomalías, o asignación variable de bits a diferentes partes de la imagen hiperespectral comprimida.

Los resultados obtenidos con VQPCA mejoran los existentes con PCA como reductor dimensional, si bien las mejoras requieren de un cuidadoso ajuste de los parámetros, que depende de la imagen concreta que se esté comprimiendo.

Posteriormente se ha realizado un análisis de la ejecución, determinando cuál de las etapas del algoritmo es la más costosa en tiempo. Este premio se lo lleva, con diferencia, la parte de

codificación, que pese a su sencillez aritmética, requiere de numerosos bucles a lo largo de la imagen para poder ejecutarse por completo. Es precisamente esta sencillez aritmética la que hace a la codificación candidata ideal para FPGA.

Se ha implementado el codificador de bloques del estándar JPEG2000, utilizado dentro del algoritmo, en VHDL, comprobando su validez contra el desarrollo software.

Los resultados muestran una clara mejora al ejecutarse en una FPGA último modelo (Virtex-7). También se consiguen mejoras sobre placas resistentes a radiación como la Virtex-4, lo cual permitiría una compresión más rápida sobre satélites, también capaces de tomar imágenes hiperespectrales y por lo general bastante limitados en cuanto a memoria.

Existe además la posibilidad, por diseño, de paralelizar el algoritmo, con varios codificadores trabajando simultáneamente. Los resultados muestran una mejora aún mayor sobre la Virtex-7, desplazando en su totalidad el cuello de botella de la codificación hacia la reducción dimensional.

Todo ello además con un consumo energético prácticamente negligible si lo comparamos con un procesador tradicional, donde hablaríamos de un par de órdenes de magnitud más.

Queda abierta la vía de probar otras técnicas y opciones de compresión, fácilmente introducibles en el framework software desarrollado. Incluso la variabilidad de la calidad de la compresión en función de los parámetros haría muy interesante un trabajo que intentase optimizar su elección para imágenes concretas.

En cuanto a la implementación sobre FPGA, hay bastantes ideas en la literatura para mejorar el codificador MQ, como la segmentación o procesamiento paralelo de varios símbolos. Sin duda introducirlas en la combinación EBCoder + MQCoder aquí desarrollada mejoraría los resultados obtenidos.

Por último, numerosas partes del algoritmo aún se ejecutan en software. Su traslado a FPGA podría acelerar aún más los resultados, aumentando el margen disponible entre velocidad de compresión y captura.

Capítulo 10

Conclusions

Hyperspectral images find their most useful applications in remotely studying different subjects, with no physical interaction aside from illumination.

Captured data shows, for each pixel, the quantity of light received on all spectral bands. From infrared to ultraviolet, all of these samples allow for great amounts of information to be analyzed, and also bloat image sizes to the GigaByte territory.

Compression of huge datasets is often necessary because of storage limitations. Specially in scientific studies, where many samples need to be stored before making an in-depth analysis of the whole dataset.

Restrictions in storage are less tight in servers and data warehouses, but often arise in intermediate buffers such as the existing memory layers between sensors and the final destination of the data. A bottleneck in any of those can imply an idle sensor while data is funneled to storage, losing potential windows of capture time.

But compression does not fully solve the problem if it is not done fast, since new bottlenecks would arise.

FPGAs are a great ally in these issues. Reconfigurable chips with which to accelerate compression algorithms up to real-time speeds, also able to be sandwiched between sensors and memory removing any bottlenecks in one blow.

This work started by analyzing the most used compression techniques, from the beginning of information theory to the newest algorithms like JPEG2000. We've seen their application in hyperspectral image compression, finding great results in the literature when combining traditional image compression algorithms with dimensionality reduction.

With JPEG2000+PCA as a baseline, numerous variations of the algorithm have been tried, notably the possibility of using different dimensionality reduction algorithms, anomaly detection, and variable bit depth for different hyperspectral bands.

Results show an improvement when using VQPCA over PCA, with better improvements achieved when carefully adjusting compression parameters, which are also show to be partially image dependant.

An execution analysis was also done to pinpoint the most costly steps of the algorithm. After improving dimensionality reduction with subsampling techniques, the slowest part was determined to be the coding. It requires numerous loops over the data to be fully executed, and takes more than half the compression time despite its simple arithmetic and logic operations. This simplicity is exactly why FPGAs are the ideal platform for its acceleration.

A VHDL implementation of the block coder from the JPEG2000 standard is proposed, testing its functionality against its software counterpart.

Results show a clear improvement when executing even over legacy radiation hardened boards such as the Virtex-4QV, which would allow faster compression on satellites, a source of hyperspectral imagery and in general limited in memory resources.

An even greater improvement is show over one of the latests FPGAs, the Virtex-7. The nature of the algorithm also allows for great parallelization, with multiple coders working simultaneously. In this context, the bottleneck is shifted back to dimensionality reduction, which could be a great step to improve in the future.

All of this done with a power consumption orders of magnitude lower than a traditional processor.

Future work will include testing other techniques and compression options, which will be easily introduced in the software framework presented. Since quality is very dependent on parameters, automatically optimizing which options to use would also be very interesting to further improve compression.

Regarding the FPGA implementation, there are plenty of room for improvement in the MQ coder step, like using segmentation or processing multiple symbols in parallel. Without a doubt their inclusion on this MQCoder + EBCoder combination would further improve our results.

Lastly, almost all of the steps of the algorithm are still not executed on FPGAs. Bringing them to reconfigurable hardware could broaden the gap now available between compression and capture speed.

Bibliografía

[1] John Dalton. *A new system of chemical philosophy*. Vol. 1. Cambridge University Press, 2010.

[2] Q lieb in. *Buzkiy Gard*. CC BY-SA 4.0. URL: goo.gl/tfmpXw (visitado 16-01-2018).

[3] Tohaomg. *Tritanopia sight*. CC BY-SA 4.0. URL: commons.wikimedia.org/wiki/File:Tritanopia_sight.jpg (visitado 16-01-2018).

[4] Tohaomg. *Deuteranopia sight*. CC BY-SA 4.0. URL: commons.wikimedia.org/wiki/File:Deuteranopia_sight.jpg (visitado 16-01-2018).

[5] Tohaomg. *Monochromacy sight*. CC BY-SA 4.0. URL: commons.wikimedia.org/wiki/File:Monochromacy_sight.jpg (visitado 16-01-2018).

[6] John Dalton. *Extraordinary facts relating to the vision of colours: with observations*. Cadell y Davies, London, 1794.

[7] Gabriele Jordan y col. «The dimensionality of color vision in carriers of anomalous trichromacy». En: *Journal of vision* 10.8 (2010), págs. 12-12.

[8] Warrickball. *Dwarf star spectrat*. CC BY-SA 4.0. URL: commons.wikimedia.org/wiki/File:Dwarf_star_spectra_(luminosity_class_V)_from_Pickles_1998.png (visitado 16-01-2018).

[9] George Frederick Barker. *Memoir of Henry Draper; 1837-1882*. 1888.

[10] John Hannavy. *Encyclopedia of nineteenth-century photography*. Routledge, 2013.

[11] James Clerk Maxwell. «On the theory of three primary colours». En: Royal Institution of Great Britain. 1861.

[12] *La révélation des couleurs*. URL: http://www.autochromes.culture.fr/ (visitado 26-02-2018).

[13] Steven J Sasson y Robert G Hills. *Electronic still camera utilizing image compression and digital storage*. US Patent 5,016,107. 1991.

[14] Academia Testo. *Historia de la cámara termográfica*. URL: http://www.academiatesto.com.ar/cms/historia-de-la-camara-termografica (visitado 26-02-2018).

[15] NASA JPL. *Airborne Visible / Infrared Imaging Spectrometer*. URL: aviris-ng.jpl.nasa.gov/ (visitado 29-01-2018).

[16] Hamed Akbari y col. «Hyperspectral imaging and quantitative analysis for prostate cancer detection». En: *Journal of biomedical optics* 17.7 (2012), págs. 0760051-07600510.

[17] Driss Haboudane y col. «Hyperspectral vegetation indices and novel algorithms for predicting green LAI of crop canopies: Modeling and validation in the context of precision agriculture». En: *Remote sensing of environment* 90.3 (2004), págs. 337-352.

[18] Gustavo Camps-Valls y col. «Advances in hyperspectral image classification: Earth monitoring with statistical learning methods». En: *IEEE Signal Processing Magazine* 31.1 (2014), págs. 45-54.

[19] Consultative Committee for Space Data Systems. *CCSDS 123.0-B-1: Lossless Multispectral & Hyperspectral Image Compression*. Consultative Committee for Space Data Systems, 2015.

[20] Qian Du y James E Fowler. «Hyperspectral image compression using JPEG2000 and principal component analysis». En: *IEEE Geoscience and Remote Sensing Letters* 4.2 (2007), págs. 201-205.

[21] Comité JPEG. *Overview of JPEG*. URL: `jpeg.org/jpeg/index.html` (visitado 11-12-2017).

[22] Claude E Shannon. «A mathematical theory of communication». En: *ACM SIGMOBILE Mobile Computing and Communications Review* 5.1 (2001), págs. 3-55.

[23] August Albert Sardinas y George W Patterson. «A necessary and sufficient condition for unique decomposition of coded messages». En: *PROCEEDINGS OF THE INSTITUTE OF RADIO ENGINEERS*. Vol. 41. 3. 1953, págs. 425-425.

[24] Jean Berstel. *Combinatorics on Words: Christoffel words and repetitions in words*. Vol. 27. American Mathematical Soc., 2009.

[25] David A Huffman. «A method for the construction of minimum-redundancy codes». En: *Proceedings of the IRE* 40.9 (1952), págs. 1098-1101.

[26] Norman Abramson. «Information theory and coding». En: (1963).

[27] Patrick Billingsley. «On the coding theorem for the noiseless channel». En: *The Annals of Mathematical Statistics* 32.2 (1961), págs. 594-601.

[28] Ian H Witten, Radford M Neal y John G Cleary. «Arithmetic coding for data compression». En: *Communications of the ACM* 30.6 (1987), págs. 520-540.

[29] David Taubman y Michael Marcellin. *JPEG2000 image compression fundamentals, standards and practice*. Vol. 642. Springer Science & Business Media, 2012.

[30] Laurens Van Der Maaten, Eric Postma y Jaap Van den Herik. «Dimensionality reduction: a comparative». En: *J Mach Learn Res* 10 (2009), págs. 66-71.

[31] Hua Yu y Jie Yang. «A direct LDA algorithm for high-dimensional data—with application to face recognition». En: *Pattern recognition* 34.10 (2001), págs. 2067-2070.

[32] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[33] Hervé Abdi y Lynne J Williams. «Principal component analysis». En: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), págs. 433-459.

[34] Harold Hotelling. «Analysis of a complex of statistical variables into principal components.» En: *Journal of educational psychology* 24.6 (1933), pág. 417.

[35] Jonathon Shlens. «A tutorial on principal component analysis». En: *arXiv preprint arXiv:1404.1100* (2014).

[36] Thomas M Cover y Joy A Thomas. «Entropy, relative entropy and mutual information». En: *Elements of information theory* 2 (1991), págs. 1-55.

[37] Aapo Hyvärinen, Juha Karhunen y Erkki Oja. *Independent component analysis*. Vol. 46. John Wiley & Sons, 2004.

[38] Andrew A Green y col. «A transformation for ordering multispectral data in terms of image quality with implications for noise removal». En: *IEEE Transactions on geoscience and remote sensing* 26.1 (1988), págs. 65-74.

[39] Christopher Gordon. «A generalization of the maximum noise fraction transform». En: *IEEE Transactions on geoscience and remote sensing* 38.1 (2000), págs. 608-610.

[40] Asgeir Bjorgan y Lise Lyngsnes Randeberg. «Real-time noise removal for line-scanning hyperspectral devices using a minimum noise fraction-based approach». En: *Sensors* 15.2 (2015), págs. 3362-3378.

[41] José MP Nascimento y José MB Dias. «Vertex component analysis: A fast algorithm to unmix hyperspectral data». En: *IEEE transactions on Geoscience and Remote Sensing* 43.4 (2005), págs. 898-910.

[42] Nanda Kambhatla y Todd K Leen. «Fast non-linear dimension reduction». En: *Advances in neural information processing systems*. 1994, págs. 152-159.

[43] Marianna Caserta. *Photographic processing*. CC BY-SA 4.0. URL: commons.wikimedia.org/wiki/File:Photographic_processing.jpg (visitado 12-12-2017).

[44] David Präkel. *The visual dictionary of photography*. Ava Publishing, 2010.

[45] Pierre Jarleton. *Nikon SLR-type digital cameras*. 1997-2017. URL: apphotnum.free.fr/N2BE2.html (visitado 11-12-2017).

[46] John W. Schulze. *Early Digital*. CC BY 2.0. URL: commons.wikimedia.org/wiki/File:Early_digital!.jpg (visitado 12-12-2017).

[47] International Organization for Standardization. *Information technology — Digital compression and coding of continuous-tone still images*. ISO/IEC 10918. 1999.

[48] Google. *Guetzly*. URL: github.com/google/guetzli (visitado 11-12-2017).

[49] *Barn*. Dominio público. URL: commons.wikimedia.org/wiki/File:Barn-yuv.png (visitado 12-12-2017).

[50] *DCT JPEG*. Dominio público. URL: commons.wikimedia.org/wiki/File:Dctjpeg.png (visitado 12-12-2017).

[51] John. *Comparison between JPEG, JPEG2000 and JPEGXR*. GFDL. URL: commons.wikimedia.org/wiki/File:Comparison_between_JPEG,_JPEG_2000_and_JPEG_XR.png (visitado 13-12-2017).

[52] Justin T Rucker, James E Fowler y Nicolas H Younan. «JPEG2000 coding strategies for hyperspectral data». En: *Geoscience and Remote Sensing Symposium, 2005. IGARSS'05. Proceedings. 2005 IEEE International*. Vol. 1. IEEE. 2005, 4-pp.

[53] Barbara Penna y col. «Progressive 3-D coding of hyperspectral images based on JPEG 2000». En: *IEEE Geoscience and remote sensing letters* 3.1 (2006), págs. 125-129.

[54] Emmanuel Christophe, Corinne Mailhes y Pierre Duhamel. «Hyperspectral image compression: adapting SPIHT and EZW to anisotropic 3-D wavelet coding». En: *IEEE Transactions on Image processing* 17.12 (2008), págs. 2334-2346.

[55] David Taubman. «High performance scalable image compression with EBCOT». En: *IEEE Transactions on image processing* 9.7 (2000), págs. 1158-1170.

[56] Harris geospatial solutions. *The ENVI header format*. URL: www.harrisgeospatial.com/docs/ENVIHeaderFiles.html (visitado 18-12-2017).

[57] Peter Abeles. *Efficient Java Matrix Library*. URL: ejml.org (visitado 18-12-2017).

[58] Edward Raff. «JSAT: Java Statistical Analysis Tool, a Library for Machine Learning». En: *Journal of Machine Learning Research* 18.23 (2017), págs. 1-5. URL: jmlr.org/papers/v18/16-131.html.

[59] Aapo Hyvarinen. «Fast and robust fixed-point algorithms for independent component analysis». En: *IEEE transactions on Neural Networks* 10.3 (1999), págs. 626-634.

[60] Aapo Hyvärinen y Erkki Oja. «Independent component analysis: algorithms and applications». En: *Neural networks* 13.4 (2000), págs. 411-430.

[61] Umberto Amato y col. «Experimental approach to the selection of the components in the minimum noise fraction». En: *IEEE Transactions on Geoscience and Remote Sensing* 47.1 (2009), págs. 153-160.

[62] Haifeng Li. *Statistical Machine Intelligence & Learning Engine*. URL: github.com/haifengl/smile (visitado 09-01-2018).

[63] Tapas Kanungo y col. «An efficient k-means clustering algorithm: Analysis and implementation». En: *IEEE transactions on pattern analysis and machine intelligence* 24.7 (2002), págs. 881-892.

[64] David Arthur y Sergei Vassilvitskii. «k-means++: The advantages of careful seeding». En: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial y Applied Mathematics. 2007, págs. 1027-1035.

[65] Ranjan Maitra, Anna D Peterson y Arka P Ghosh. «A systematic evaluation of different methods for initializing the K-means clustering algorithm». En: *A separability index for clustering and classification problems with applications to cluster merging and systematic evaluation of clustering algorithms* (2011), pág. 41.

[66] Albert Cohen, Ingrid Daubechies y J-C Feauveau. «Biorthogonal bases of compactly supported wavelets». En: *Communications on pure and applied mathematics* 45.5 (1992), págs. 485-560.

[67] AR Calderbank y col. «Wavelet transforms that map integers to integers». En: *Applied and computational harmonic analysis* 5.3 (1998), págs. 332-369.

[68] The Eclipse Foundation. *Eclipse IDE*. URL: www.eclipse.org/ (visitado 17-01-2018).

[69] Daniel Báscones. *Jypec*. URL: github.com/Daniel-BG/Jypec (visitado 17-01-2018).

[70] I ISO. «12232: Photography-Electronic Still Picture Cameras: Determination of ISO Speed». En: *International Organization for Standardization, Geneva, Switzerland* (1997).

[71] Zhou Wang y col. «Image quality assessment: from error visibility to structural similarity». En: *IEEE transactions on image processing* 13.4 (2004), págs. 600-612.

[72] Spectir Advanced Hyperspectral y Geospatial Solutions. *Free Data Samples*. URL: www.spectir.com/free-data-samples/ (visitado 18-01-2018).

[73] Fernando Garcia-Vilchez y col. «On the impact of lossy compression on hyperspectral image classification and unmixing». En: *IEEE Geoscience and remote sensing letters* 8.2 (2011), págs. 253-257.

[74] Xiaoli Tang, William A Pearlman y James W Modestino. «Hyperspectral image compression using three-dimensional wavelet coding». En: *Image and Video Communications and Processing 2003*. Vol. 5022. International Society for Optics y Photonics. 2003, págs. 1037-1048.

[75] Qian Du y James E Fowler. «Low-complexity principal component analysis for hyperspectral image compression». En: *The International Journal of High Performance Computing Applications* 22.4 (2008), págs. 438-448.

[76] Daniel Báscones. *Vypec*. URL: github.com/Daniel-BG/Vypec (visitado 25-01-2018).

[77] Maël Hörz. *HxD - Freeware Hex Editor and Disk Editor, 1.7.7.0*. URL: mh-nexus.de/en/hxd/ (visitado 15-01-2018).

[78] Daniel Báscones. «Implementación sobre FPGA de un algoritmo de compresión de imágenes hiperespectrales bajo el estándar CCSDS 123.0-B-1». Trabajo de fin de Grado. Universidad Complutense de Madrid, 2016.

[79] www.nandland.com. *UART, Serial Port, RS-232 Interface*. URL: www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html (visitado 15-01-2018).

[80] Xilinx. *Vivado Design Suite, 2016.3*. URL: www.xilinx.com/products/design-tools/vivado.html (visitado 15-01-2018).

[81] Realterm. *Realterm: Serial Terminal, 3.0.0.30*. URL: realterm.sourceforge.io/ (visitado 15-01-2018).

[82] Xilinx. *Xilinx Virtex-7 FPGA VC709 Connectivity Kit*. URL: www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html (visitado 15-01-2018).

[83] Xilinx. *ISE Design Suite 14.7*. URL: www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html (visitado 29-01-2018).

[84] Xilinx. *ML410 Embedded Development Platform*. URL: www.xilinx.com/support/documentation/boards_and_kits/ug085.pdf (visitado 29-01-2018).

[85] Xilinx. *Virtex 4 family overview*. URL: www.xilinx.com/support/documentation/data_sheets/ds112.pdf (visitado 23-01-2018).

[86] Xilinx. *7 Series FPGAs Data Sheet: Overview*. URL: www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (visitado 23-01-2018).

[87] Xilinx. *Space-grade Virtex-5QV FPGA*. URL: www.xilinx.com/products/silicon-devices/fpga/virtex-5qv.html (visitado 29-01-2018).

[88] Kishor Sarawadekar y Swapna Banerjee. «Low-cost, high-performance VLSI design of an MQ coder for JPEG 2000». En: *Signal Processing (ICSP), 2010 IEEE 10th International Conference on*. IEEE. 2010, págs. 397-400.

[89] Kai Liu y col. «A high performance MQ encoder architecture in JPEG2000». En: *INTEGRATION, the VLSI journal* 43.3 (2010), págs. 305-317.

[90] Nopphol Noikaew y Orachat Chitsobhuk. «Dual Symbol Processing for MQ arithmetic coder in JPEG2000». En: *Image and Signal Processing, 2008. CISP'08. Congress on*. Vol. 1. IEEE. 2008, págs. 521-524.

[91] Chung-Jr Lian y col. «Analysis and architecture design of block-coding engine for EBCOT in JPEG 2000». En: *IEEE Transactions on circuits and systems for video technology* 13.3 (2003), págs. 219-230.

[92] Yun-Tai Hsiao y col. «High-speed memory-saving architecture for the embedded block coding in JPEG2000». En: *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*. Vol. 5. IEEE. 2002, págs. V-V.

Glosario

Símbolo	Descripción
b	Número de bits utilizados para cuantizar en la compresión
baldosa	Región rectangular dentro de una imagen
banda	Una banda de una imagen hiperespectral contiene todas las muestras correspondientes a una longitud de onda concreta. Tiene las dimensiones espaciales de la imagen original, pero es unidimensional en el espectro
BIL	Intercalado por línea y banda
BIP	Intercalado por píxel y banda
bloque	Sección rectangular de una banda, con un tamaño máximo de 4096 muestras. Puede abarcar varias subbandas
BSQ	Intercalado secuencial por banda
codificación	Proceso mediante el cual se transforma y comprime un conjunto de datos en un flujo de bits, que posteriormente puede ser descodificado. La codificación no conlleva pérdida
compresión	Proceso mediante el cual se reduce el tamaño de un conjunto de datos. Puede hacerse con y sin pérdida
contexto	Tipo enumerado que representa el entorno de un bit a la hora de ser codificado
DCT	Transformada Discreta del Coseno
EBCoder	Codificador de bloque utilizado en JPEG2000. Recorre bloques de una imagen de hasta 64×64 píxeles bit a bit, utilizando el MQCoder para codificarlos
entropía	Cantidad de información media que genera una fuente aleatoria. Se determina en función de la probabilidad de que la fuente genere diferentes símbolos
f	Función que realiza la reducción dimensional
FPGA	Field Programmable Gate Array
frame	Una *frame* o cuadro de una imagen hiperespectral es un corte vertical (es decir, una línea de píxeles) de la imagen
g	Función inversa a f que realiza un aumento dimensional
ICA	Independent Component Analysis
JPEG	Joint Photographic Experts Group
línea	Una *línea* de píxeles en una imagen hiperespectral es sinónimo de un frame. Se utiliza en su lugar al hablar de las dimensiones de una imagen (bandas×líneas×muestras)

Símbolo	Descripción
m	Dimensión del espacio transformado en los algoritmos de reducción dimensional
MNF	Minimum Noise Fraction
MQCoder	Codificador aritmético binario utilizado en la codificación de JPEG2000
muestra	Una muestra de una imagen hiperespectral es un valor concreto en unas coordenadas dadas. Habitualmente son números enteros de 16 bits, y pueden o no tener signo. En el contexto de la codificación, se usará el término muestra indistintamente para referir tanto a los 16 bits, como a bits individuales. También utilizamos muestras para referirnos al ancho de la imagen hiperespectral
n	Número de bandas que tiene una imagen hiperespectral, o dimensión de partida en los algoritmos de reducción dimensional
p	Número de píxeles en una imagen, igual al producto de sus dimensiones espaciales
PCA	Principal Component Analysis
píxel	Un píxel de una imagen contiene, para una coordenada espacial, todas las muestras de las diferentes bandas de la imagen en ese punto. En una fotografía normal, un píxel está formado por una tripleta (r, g, b) de colores rojo, verde y azul. En una imagen hiperespectral es un vector de n componentes
plano	Conjunto de los bits enésimos de todas las muestras de un bloque. Cada bloque se codifica progresivamente de plano más a menos significativo. En caso de tratar con muestras con signo, existe un plano de signo que indica para cada muestra el signo de su valor
r	Número de dimensiones a las que reducen los reductores dimensionales, sinónimo a m
RGB	Red Green Blue
S	Matriz de covarianza de los vectores de X, de tamaño $\mathcal{M}_{n\times n}$
significancia	Tipo enumerado utilizado para indicar el estado de las muestras al realizar la codificación en bloque
SNR	Ratio de Señal a Ruido
subbanda	Sección rectangular de una banda, donde todas las muestras corresponden al mismo sector de la pasada de ondícula. Existen 4 tipos de subbanda: LL, HL, LH, HH según sean resultantes del paso bajo (L) o alto (H) de la ondícula en horizontal y vertical respectivamente
SVD	Singular Value Decomposition
t	Elemento del vector $\boldsymbol{t}$, o muestra de la imagen reducida. También porcentaje de píxeles utilizados para el submuestreo a la hora de hacer el entrenamiento de los reductores dimensionales
UART	Universal Asynchronous Receiver Transmitter
VCA	Vector Component Analysis
VQPCA	Vector Quantization Principal Component Analysis

Símbolo	Descripción
W	Matriz o conjunto de vectores utilizados para la reducción dimensional mediante proyección, que se hace mediante el producto $XW = T$. Tiene tamaño $\mathcal{M}_{n \times m}$. Símbolo también utilizado para denotar una transformación de ondícula sobre una serie
$\bar{W}$	Matriz o conjunto de vectores utilizado para el aumento dimensional contrario a la reducción obtenida con W. En muchas ocasiones se cumplirá $\bar{W} = W^T$
x	Elemento del vector $\boldsymbol{x}$, equivalente a una muestra de la imagen
$\bar{\boldsymbol{x}}$	Media del conjunto X
YCbCr	Componentes de color Y (luminancia o brillo), Cb (Crominancia azul o cantidad de azul), Cr (Crominancia roja o cantidad de rojo)
Z	Conjunto resultante de restar la media $\bar{\boldsymbol{x}}$ al conjunto X

Índice alfabético